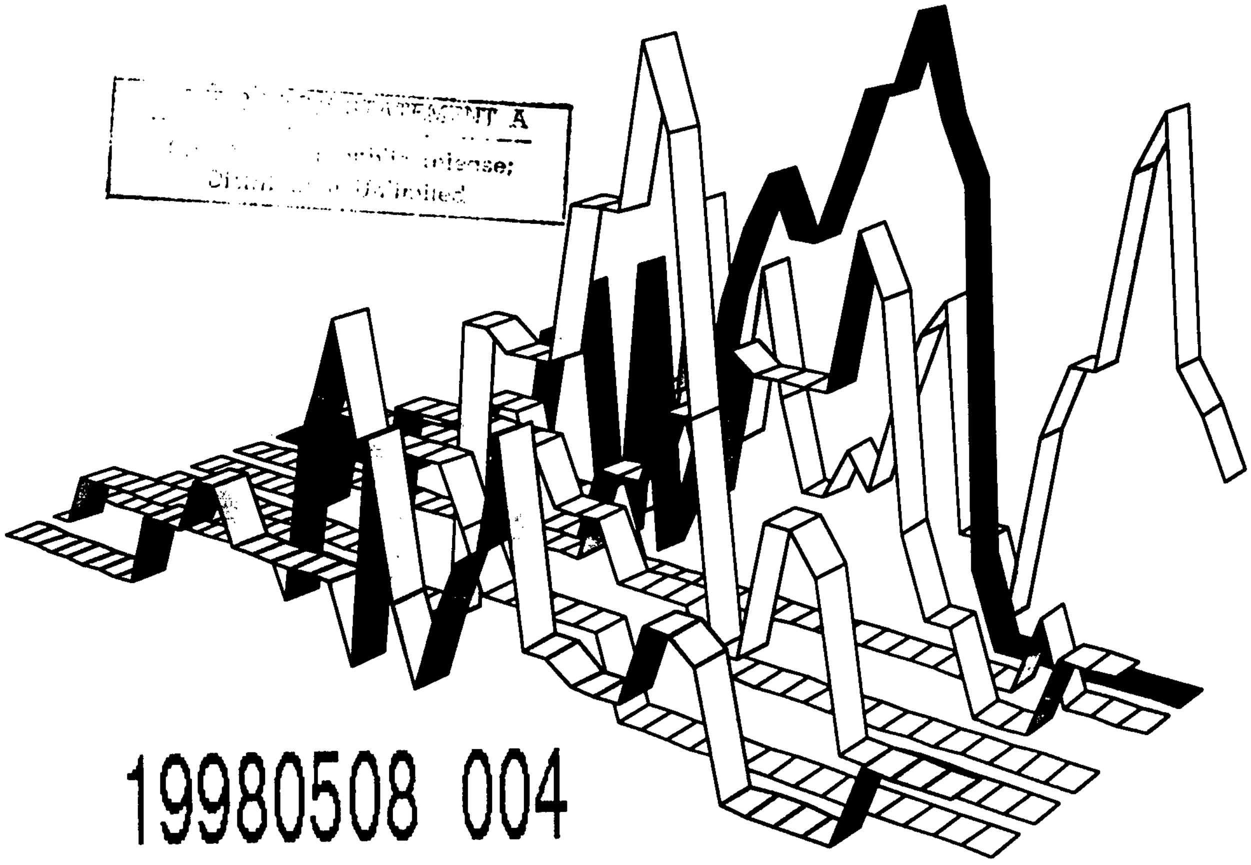
Language Definition DSDL for Hawk

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.4]

Prepared for:
USAF
Electronic Systems Center/AVK



OREGON
GRADUATE
INSTITUTE OF
SCIENCE &
TECHNOLOGY

Language Definition DSDL for Hawk

Pacific Software Research Center
April 22, 1998

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.4]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared for:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Microprocessor Specification in Hawk*

John Matthews johnm@cse.ogi.edu
Byron Cook byron@cse.ogi.edu
John Launchbury jl@cse.ogi.edu

Abstract

Modern microprocessors require an immense investment of time and effort to create and verify, from the high-level architectural design downwards. We are exploring ways to increase the productivity of design engineers by creating a domain-specific language for specifying and simulating processor architectures. We believe that the structuring principles used in modern functional programming languages, such as static typing, parametric polymorphism, first-class functions, and lazy evaluation provide a good formalism for such a domain-specific language, and have made initial progress by creating a library on top of the functional language Haskell. We have specified the integer subset of an out-of-order, superscalar DLX microprocessor, with register-renaming, a reorder buffer, a global reservation station, multiple execution units, and speculative branch execution. Two key abstractions of this library are the signal abstract data type (ADT), which models the simulation history of a wire, and the transaction ADT, which models the state of an entire instruction as it travels through the microprocessor.

1 Introduction

Modern microprocessor technologies have substantially increased processor performance. For example, *pipelining* allows a processor to overlap the execution of several instructions at once. With *superscalar* execution, multiple instructions are read per clock cycle. *Out-of-order execution*, where some instructions that logically come after a given instruction may be executed before the given instruction, can also greatly increase processor speed [6]. All of these technologies dramatically increase design complexity. In fact, creating and verifying these designs is a significant proportion of the total microprocessor development lifecycle. As the number of possible gates in future microprocessors increases exponentially, so too does design complexity.

At OGI, we have developed the *Hawk* language for building executable specifications of microprocessors, concentrating on the level of micro-architecture. In the long term we plan for Hawk to be a standalone language. In the meantime we have embedded our language into Haskell, a strongly-typed functional language with lazy (demand-driven) evaluation, first-class functions, and parametric polymorphism [5] [12].

The library makes essential use of these features. As an example, we have used Hawk to specify and simulate the integer portion of a pipelined DLX microprocessor[4]. The DLX is a complete microprocessor and is a widely used model among researchers. Several DLX simulators exist, as well as a version of the Gnu C compiler that generates DLX assembly instructions. The processor includes the most common instructions found in commercial RISC processors. Our specification, including data and control hazard resolution, is only two pages of Hawk code. A non-pipelined version of the processor was specified in half of a page.

In this report, we introduce the concepts behind Hawk. Rather than attempting a detailed explanation of the whole of the DLX with all of its inherent complexity, we have chosen to exhibit the techniques on a considerably simplified model. A corresponding annotated specification of the DLX itself can be found in [13].

2 The Hawk Library

We start with a simple example that introduces several functions used in later examples. Consider the resettable counter circuit of Figure 1.

The *reset* wire is Boolean valued, while the other

*Copyright 1998 IEEE. Published in the Proceedings of ICCL'98, May 1998 Chicago, Illinois. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

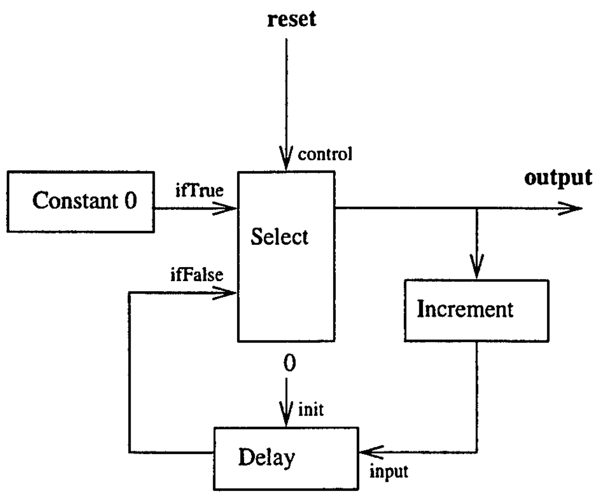


Figure 1: **Resettable Counter. A simple circuit that counts the number of clock cycles between reset signals.**

wires are integer valued. Of course, in silicon, integer-valued wires are represented by a vector of Boolean wires, but as a design abstraction, a Hawk user may choose to use a single wire. The circuit counts (and outputs) the number of clock cycles since *reset* was last asserted.

2.1 Signals

Notice that there is no explicit clock in the diagram. Rather, each wire in the diagram carries a *signal* (integer or boolean valued) which is an implicitly clocked value. The output of a circuit only changes between clock cycles. We build signals using an abstract type constructor called `Signal`. As a mental model we could think of a value of type `Signal a` as a function from integers to values of type `a`.

```
type Signal a = (Int -> a)
```

The integers denote the current time, measured as the number of clock cycles since the start of the simulation. Circuits and components of circuits are represented as functions from signals to signals. This view of signals is used extensively in the hardware verification community [9] [14]. Equivalently, we can think of signals as infinite sequences of values.

In the resettable counter example above, the *constant 0* circuit outputs zero on every clock cycle. The *select* component chooses between its inputs on each clock cycle depending on the value of *reset.* If *reset* is asserted on a given cycle (has value *true*), then the output is equal to *select*'s top input, in this case zero. If *reset* is not asserted, then its output is the value of its bottom input. In either case, *select*'s output is the output of the entire circuit, as well as the input to the *increment* component, which simply adds 1 to its input. The output of *increment* is fed into the *delay* component. A delay component outputs whatever was on its input in the previous clock cycle: it "delays" its input by one cycle. However, on the first clock cycle of the simulation there is no previous input, so on the first cycle *delay* outputs whatever is on its *init* input, which is zero in this circuit.

2.2 Components

The components used in the resettable counter are trivial examples of the sorts of things provided by the Hawk library, but let's look at a specification of each component in turn.

The simplest component is `constant`

```
constant :: a -> Signal a
```

The constant function takes an input of any type `a`, and returns an output of type `Signal a`, that is, a sequence of values of type `a`. For every clock cycle, `(constant x)` always has the same value `x`.

The next component is `select`:

```
select :: Signal Bool ->
          Signal a ->
          Signal a ->
          Signal a
```

This declares `select` to be a function. In a Hawk declaration, anything to the left of an arrow is a function argument. Thus, the expression `(select bs xs ys)`, where `bs` is a Boolean signal, and `xs` and `ys` are signals of type `a`, will return an output signal of type `a`. The values of the output signal are drawn from `xs` and `ys`, decided each clock tick by the corresponding value of `bs`. For example, if

```
bs = <True,False,True,False,...>,
xs = <x1,x2,x3,x4,...>,
ys = <y1,y2,y3,y4,...>
```

then `(select bs xs ys)` is equal to the signal `<x1,y2,x3,y4,...>`.

Hawk treats functions as first-class values, allowing them to be passed as arguments to other functions or returned as results. First-class functions allow us to specify a generic `lift` primitive, which "lifts" a normal function from type `a` to type `b` into a function over the corresponding signal types:

```
lift :: (a -> b) -> Signal a -> Signal b
```

The expression (lift f xs), where xs = <x1,x2,x3,...>, is equal to the signal <f x1, f x2, f x3, ...>.

The increment component is defined in terms of lift:

```
increment :: Signal Int -> Signal Int
increment xs = lift (+ 1) xs
```

Given the xs input signal, increment adds one to each component of xs and returns the result.

The delay component is more interesting:

```
delay :: a -> Signal a -> Signal a
```

This function takes an initial value of type a, and an input signal of type Signal a, and returns a value of type Signal a (the input arguments are in reverse order from the diagram). At clock cycle zero, the expression (delay initVal xs) returns initVal. Otherwise the expression returns whatever value xs had at the previous clock cycle. This function can thus propagate values from one clock cycle to the next. Note that delay is polymorphic, and can be used to delay signals of any type.

2.3 Using the components

Once we have defined primitive signal components like the ones above, we can define the resettable counter:

```
resetCounter :: Signal Bool -> Signal Int
resetCounter reset = output
  where
    output =
      select reset
              (constant 0)
              (delay 0 (increment output))
```

The resetCounter definition takes reset as a Boolean signal, and returns an integer signal. The reset signal is passed into select. On every clock cycle where reset returns True, select outputs 0, otherwise it outputs the result of the delay function. On the first clock cycle delay outputs 0, and thereafter outputs the result of whatever (increment output) was on the previous clock cycle. The output of the whole circuit is the output of the select function, here called output. Notice that output is used twice in this function: once as the input to increment, and once as the result of the entire function. This corresponds to the fact that the output wire in Figure 1 is split and used in two places. Whenever a wire is duplicated in this fashion, we must use a where statement in Hawk to name the wire.

2.4 Recursive Definitions

There is something else curious about the output variable. It is being used recursively in the same place it is being defined! Most languages only allow such recursion for functions with explicit arguments. In Hawk, one can also define recursive data-structures and functions with implicit arguments, such as the one above.

If we didn't have this ability, we would have had to define resetCounter as follows:

```
resetCounter reset = output
  where
    output time =
      (select reset
              (constant 0)
              (delay 0 (increment output))) time
```

Every time we have a cycle in a circuit, we have to create a local recursive function, passing an explicit time parameter. This breaks the abstraction of the Signal ADT. In fact, in the real implementation of signals, we don't use functions at all. We use infinite lists instead. Each element of the list corresponds to a value at a particular clock cycle; the first list element corresponds to the first clock cycle, the second element to the second clock cycle, and so on. By storing signals as lazy lists, we compute a signal value at a given clock cycle only once, no matter how many times it is subsequently accessed.

Haskell allows recursive definitions of abstract data structures because it is a lazy language, that is, it only computes a part of a data structure when some client code demands its value. It is lazy evaluation that allows Haskell to simulate infinite data structures, such as infinite lists.

3 A Simple Microprocessor

As we noted in the introduction, the DLX architecture is too complex to explain in fine detail in an introductory report. Thus for pedagogical purposes we show how to use similar techniques to specify a simple microprocessor called SHAM (Simple HAwk Microprocessor). We begin with the simplest possible SHAM architecture (unpipelined), and then add features: pipelining, and a memory-cache.

The unpipelined SHAM diagram is shown in Figure 2. The microprocessor consists of an ALU and a register file. The ALU recognizes three operations: ADD, SUB, and INC. The ADD and SUB operations add and subtract, respectively, the contents of the two ALU inputs. The INC operation causes the ALU to increment its first input by one and output the result.

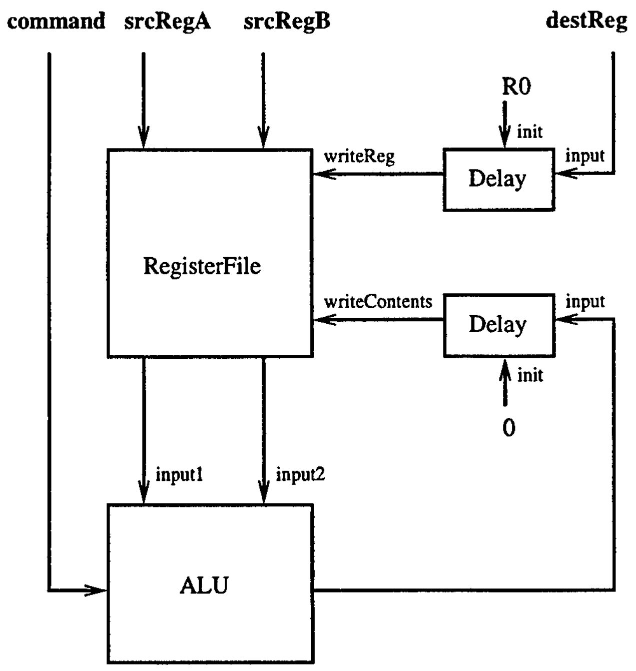


Figure 2: **Unpipelined version of SHAM.**

The register file contains eight integer registers, numbered R0 through R7. Register R0 is hardwired to the value zero, so writes to R0 have no effect. The register file has one write-port and two read-ports. The write-port is a pair of wires; the register to update, called *writeReg*, and the value being written, called *writeContents*. The input to each read-port is a wire carrying a register name. The contents of the named read-port registers are output every cycle along the wires *contentsA* and *contentsB*. If a register is written to and read from during the same clock cycle, the newly written value is reflected in the read-port's output. This is consistent with the behavior of most modern microprocessor register files.

SHAM instructions are provided externally; in our drive for simplicity there is no notion of a program counter. Each instruction consists of an ALU operation, the destination register name, and the two source register names. For each instruction the contents of the two source registers are loaded into the ALU's inputs, and the ALU's result is written back into the destination register.

3.1 Unpipelined SHAM Specification

Let us assume we have already specified the register file and ALU, with the signatures below:

```
data Reg = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7
```

```
regFile :: (Signal Reg, Signal Int) ->
           Signal Reg ->
           Signal Reg ->
           (Signal Int, Signal Int)

data Cmd = ADD | SUB | INC

alu :: Signal Cmd -> Signal Int -> Signal Int ->
       Signal Int
```

The `regFile` specification takes a write-port input, two read-port inputs, and returns the corresponding read-port outputs. The `alu` specification takes a command signal and two input signals, and returns a result signal. Given these signatures and the previous definition of `delay`, it is easy in Hawk to specify an unpipelined version of SHAM:

```
sham1 :: (Signal Cmd,Signal Reg,
          Signal Reg,Signal Reg) ->
         (Signal Reg,Signal Int)

sham1 (cmd,destReg,srcRegA,srcRegB) =
  (destReg',aluOutput')
  where
    (aluInputA,aluInputB) =
      regFile (destReg',aluOutput')
              srcRegA srcRegB
    aluOutput = alu cmd aluInputA aluInputB
    aluOutput' = delay 0 aluOutput
    destReg' = delay R0 destReg
```

The definition of `sham1` takes a tuple of signals representing the stream of instructions, and returns a pair of signals representing the sequence of register assignments generated by the instructions. The first three lines in the body of `sham1` read the source register values from the register file and perform the ALU operation. The next two lines delay the destination register name and ALU output, in effect returning the values of the previous clock cycle. The delayed signals become the write-port for the register file. It is necessary to delay the write-port since modifications to the register file logically take effect for the next instruction, not the current one.

3.2 Pipelining

Suppose we wanted to increase SHAM's performance by doubling the clock frequency. We will assume that, while `sham1` could perform both the register file and ALU operations within one clock cycle, with the increased frequency it will take two clock cycles to perform both functions serially. We use pipelining to increase the overall performance. While the ALU is working on instruction n, the register file will be writing the result of instruction $n-1$ back into the appropriate register, and simultaneously reading the source registers of instruction $n+1$.

But now consider the following sequence of instructions, such as:

```
R2 <- R1 ADD R3
R4 <- R2 SUB R5
```

When the ADD instruction is in the ALU stage, the SUB instruction is in the register-fetch stage. But one of the registers that is being fetched (R2), has not been written back into the register file yet, because the ALU is still calculating the result. The SUB instruction will read an out-of-date value for R2. This is an example of a *data hazard*, where naive pipelining can produce a result different from the unpipelined version of a microprocessor. To resolve this hazard, we will first add *bypass logic* to the pipeline, then later abstract away from this added inconvenience.

Figure 3 contains the diagram of a pipelined version of SHAM with bypass logic. By the time the source operands to the SUB instruction (R2 and R5) are ready to be input into the ALU, the up-to-date value for R2 is stored in the delay circuit between the ALU and the register file's write-port. The bypass logic uses this stored value of R2 as the input to the ALU, rather than the out-of-date value read from the register file. The bypass logic examines the incoming instructions to determine when this is necessary. The following code contains the Hawk specification:

```
sham2 :: (Signal Cmd,Signal Reg,
          Signal Reg,Signal Reg)
         ->
         (Signal Reg,Signal Int)

sham2 (cmd,destReg,srcRegA,srcRegB) =
  (destReg'',aluOut')
  where
    (valueA,valueB) = regFile (destReg'',aluOut')
                              srcRegA srcRegB

    valueA'   = delay 0 valueA
    valueB'   = delay 0 valueB
    destReg'  = delay R0 destReg
    cmd'      = delay ADD cmd

    aluInputA  = select validA valueA' aluOut'
    aluInputB  = select validB valueB' aluOut'

    aluOut     = alu cmd' aluInputA aluInputB

    aluOut'    = delay 0 aluOut
    destReg''  = delay R0 destReg'

    --- Control logic ---

    validA     = delay True (noHazard srcRegA)
    validB     = delay True (noHazard srcRegB)

    noHazard :: Signal Reg -> Signal Bool
    noHazard srcReg =
      sigOr (sigEqual destReg' (constant R0))
            (sigNotEqual destReg' srcReg)
```

The first two lines after the `where` keyword read the contents of the source registers from the register file. The next four lines delay the source register contents, the ALU command, and the destination register name by one cycle. The two `select` commands decide whether the delayed values should be bypassed. The decision is made by the Boolean signals `validA` and `validB`, which are defined in the control logic section. The next line performs the ALU operation. The last two lines in the data-flow section delay the ALU result and the destination register. The delayed result, called `aluOut'`, is written back into the register file in the register named by `destReg''`, as indicated in the first two lines of the section.

The control logic section determines when to bypass the ALU inputs. The signals `validA` and `validB` are set to `True` whenever the corresponding ALU input is up-to-date. The definition of these signals uses the function `noHazard`, which tests whether the previous instruction's destination register name matches a source register name of the current instruction. If they do, then the function returns `False`. The exception to this is when the destination register is R0. In this case the ALU input is always up-to-date, so `noHazard` returns `True`.

3.3 Transactions

The definition of sham2 highlights a difficulty of many such specifications. Although the data flow section is relatively easy to understand, the control logic section is far from satisfactory. In fact, it often takes nearly as many lines of Hawk code to specify the control logic as it does to specify the data flow, and mistakes in the control logic may not be easy to spot. We need a more intuitive way of defining control logic sections in microprocessors.

We use a notion of *transactions* within Hawk to specify the state of an entire instruction as it travels through the microprocessor (similar in spirit to

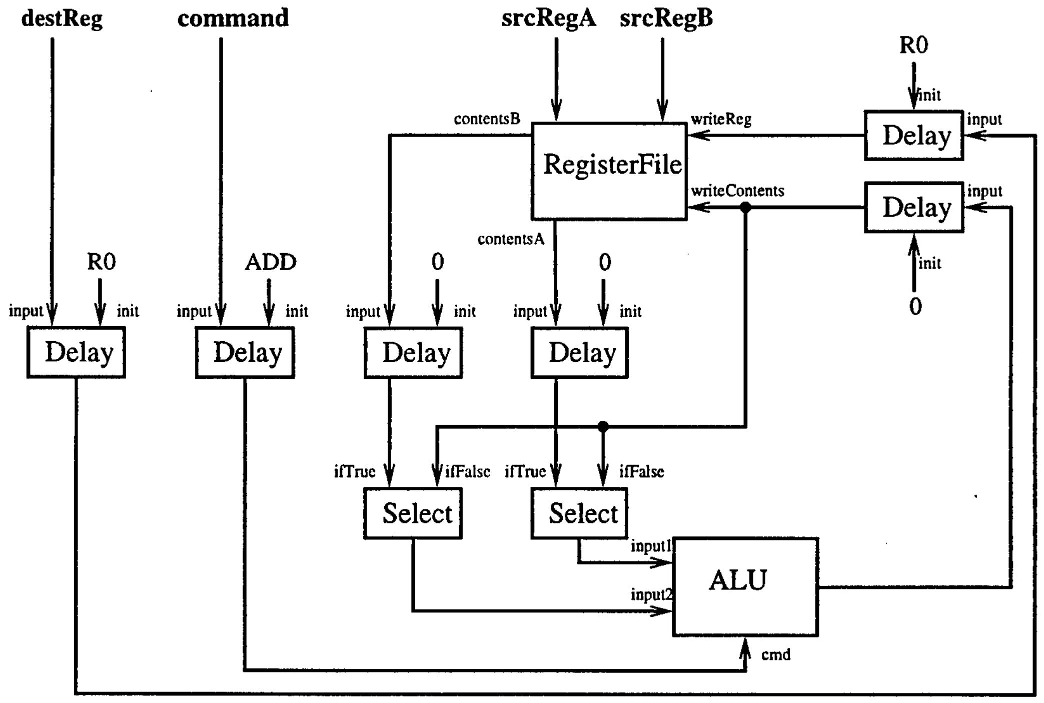


Figure 3: **Pipelined SHAM. Since the register file and the ALU each now take one clock cycle to complete, we now need *Delay* circuits. The *Delay* circuits in turn require us to add *Select* circuits to act as bypasses. The logic controlling the *Select* circuits is not shown.**

Aagaard and Leeser [1]). A transaction holds an instruction's source operand values, the ALU command, and the destination operand value. Transactions also record the register names associated with the source and destination operands:

```
data Transaction = Trans DestOp Cmd [SrcOp]

type DestOp  = Operand
type SrcOp   = Operand
type Operand = (Reg,Value)

data Value = Unknown | Val Int
```

An operand is a pair containing a register and its value. Values can either be "unknown" or they can be known, e.g. `Val 7`.

For example, the instruction (`R3 <- R2 ADD R1`), when it has completed, would be encoded as shown below (assume that register `R2` holds the value 3, and `R1` holds 4):

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

This expression states that register `R3` should be assigned the value 7 as a result of adding the contents of register `R2` and `R1`.

Not all of the register values in a transaction are known in the early stages of the pipeline. When a register name does not have an associated value yet, it is assigned the value `Unknown`. For example, if the above instruction had not reached the ALU stage yet, then the corresponding transaction would be:

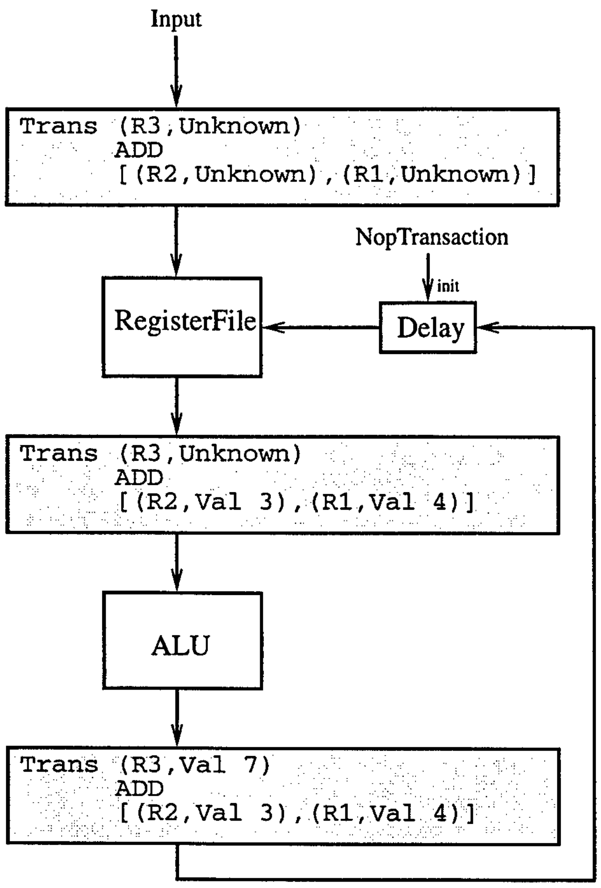
```
Trans (R3,Unknown) ADD [(R2,(Val 3)),(R1,Val 4))]
```

Figure 4 shows how a transaction's values are filled in as it flows through the pipeline.

3.4 Transaction structure

In general, the `Transaction` datatype contains three subfields. The first field holds the destination register name and its current state. The *state* of a register indicates the current value for the register at a given stage of the pipeline. Possible state values are `Unknown`, or (`Val k`). The second field is the instruction's ALU operation, in this case the `ADD` command. The third field holds a list of source operand register names and their corresponding states. In this example, it holds the names and states for the source operands `R2` and `R1`.

The instruction (`R3 <- R2 ADD R1`), before it enters the SHAM pipeline, is encoded as the transaction:



Input
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
NopTransaction
init
RegisterFile
Delay
Trans (R3,Unknown) ADD [(R2,Val 3),(R1,Val 4)]
ALU
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]

Figure 4: **A transaction as it flows through the pipeline. As the transaction progresses, its operands become more refined.**

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

At this point, none of the register values are known.

3.5 Changes to handle transactions

We change the `regFile` and `alu` functions so that they take and return transactions:

```
regFile :: Signal Transaction ->
           Signal Transaction ->
           Signal Transaction

alu :: Signal Transaction ->
       Signal Transaction
```

Because the register file needs to both write new values to the CPU registers and read values from them, the `regFile` function takes a *write-transaction* and a *read-transaction* as inputs. The function examines the destination register field of the write-transaction and updates the corresponding register in the register file. It outputs the read-transaction, modified so that all of the source register fields contain current values from the register file. For example, suppose `regFile` is applied to the completed write-transaction:

```
Trans (R1,Val 4) INC [(R1,Val 3)]
```

and uses as the read transaction:

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

Further, assume that register R1 is assigned 20 and R2 is assigned 3 before `regFile`'s application. Then `regFile` will update R1 to contain 4 from the write-transaction, and will output a new transaction that is identical to the read-transaction, except that all of the source registers have been assigned current values from the register file:

```
Trans (R3,Unknown) ADD [(R2,Val 3),(R1,Val 4)]
```

The revised `alu` function takes a transaction whose source operands have values, performs the appropriate operation, and outputs a modified transaction whose destination field has been filled in. Thus if the `ADD` transaction above were given to `alu`, it would return:

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

3.6 Unpipelined SHAM

Using transactions, the unpipelined version of SHAM is even easier to specify than it was before.

```
sham1Trans :: Signal Transaction ->
              Signal Transaction
sham1Trans instr =  aluOutput'
  where
    aluInput = regFile aluOutput' instr
    aluOutput = alu aluInput
    aluOutput' = delay nop aluOutput

nop = Trans (R0,Val 0) ADD [(R0,Val 0),(R0,Val 0)]
```

But the real benefit of transactions comes from specifying more complex micro-architectures, as we shall see next.

3.7 SHAM2 with Transactions

Transactions are designed to contain the necessary information for concisely specifying control logic. The control logic needs to determine when an instruction's source operand is dependent on another instruction's destination operand. To calculate the dependency, the source and destination register names must be available. The transaction carries these names for each instruction. Because of this additional information, bypass logic is easily modeled with following combinator:

```
bypass :: Signal Transaction ->
          Signal Transaction ->
          Signal Transaction
```

The bypass function usually just outputs its first argument. Sometimes, however, the second argument's destination operand name matches one or more of the first argument's source operand names. In this case, the source operand's state values are updated to match the destination operand state value. The updated version of the first argument is then returned.

So if at clock cycle *n* the first argument to bypass is:

```
Trans (R4,Unknown) ADD [(R3,Val 12),(R2,Val 4)]
```

and the second argument at cycle *n* is:

```
Trans (R3,Val 20) SUB [(R8,Val 2),(R11,Val 10)]
```

then because R3 in the second transaction's destination field matches R3 in the first transaction's source field, the output of bypass will be an updated version of the first transaction:

```
Trans (R4,Unknown) ADD [(R3,Val 20),(R2,Val 4)]
```

One special case to bypass's functionality is when a source register is R0. Since R0 is a constant register, it does not get updated. The pipelined version of SHAM with bypass logic is now straightforward. Notice that no explicit control logic is needed, as all the decisions are taken locally in the bypass operations.

```
SHAM2Trans :: Signal Transaction ->
              Signal Transaction

SHAM2Trans instr = aluOutput'
  where
    readyInstr = regFile aluOutput' instr
    readyInstr' = delay nopTrans readyInstr
    aluInput = bypass readyInstr' aluOutput'
    aluOutput = alu aluInput
    aluOutput' = delay nopTrans aluOutput
```

The first line takes instr and fills in its source operand fields from the register file. The filled-in transaction is delayed by one cycle in the second line. In the third line bypass is invoked to ensure that all of the source operands are up-to-date. Finally the transaction result is computed by alu and delayed one cycle so that the destination operand can be written back to the register file.

3.8 Hazards

There are some microprocessor hazards that cannot be handled through bypassing. For example, suppose we extended the SHAM architecture to process load and store instructions:

```
R3 <- MEM[R2]
MEM[R5] <- R2
```

The first instruction above is a load instruction; it loads the contents of the address pointed to by R2 into R3. The second instruction is a store; it stores the contents of R2 into the address pointed to by R5. A block diagram of the extended SHAM architecture is shown in Figure 5. There is now a load/store pipeline stage after the ALU stage. However, this introduces a new problem. Suppose SHAM executes the following two instructions in sequence:

```
R2 <- MEM[R1]
R4 <- R2 ADD R3
```

These two instructions have a data hazard, just as before, but we can not use bypassing to resolve it. Bypassing depends on having a value to bypass at the *beginning* of a clock cycle, but R2's value won't be known until the end of the cycle, after the memory contents have been retrieved from the memory cache. To resolve this hazard, we have to *stall* the pipeline at the register-fetch stage. When the first instruction has reached the end of the ALU stage, the second instruction will have reached the end of the register-fetch stage. At this point the delay circuits between the register-fetch stage and the ALU stage are overridden; on the next clock cycle they instead output the equivalent of a no-op instruction. The register-fetch stage itself re-reads the second instruction on the next clock cycle. In effect, the pipeline stall inserts a no-op instruction between the two instructions involved in the hazard:

```
R2 <- MEM[R1]
NOP
R4 <- R2 ADD R3
```

Now when the ADD instruction is about to be processed by the ALU, the load instruction has already completed the memory stage. R2's value is held in the pipeline registers after the memory stage, so bypass logic can be used to bring the ALU's input up-to-date. In order to stall correctly, we have to re-read the second instruction. Thus stalling reduces the performance of the pipeline.

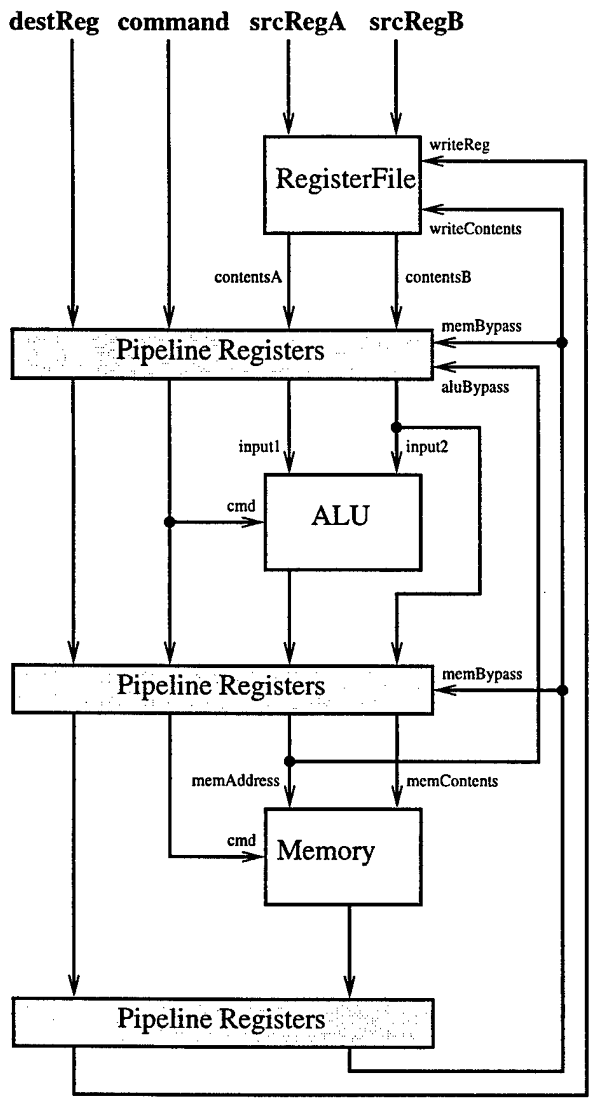


Figure 5: **Block diagram of extended SHAM pipeline. Each *Pipeline Register* circuit is made up of multiple *Delay* and *Select* circuits. The *Select* circuits are used for bypassing, ensuring that the source operands are up-to-date.**

3.9 Hawk Specification of Extended SHAM

In this section we will give more evidence of the simplifying power of transactions by specifying the extended SHAM architecture. The load/store extension significantly complicates the control logic for the SHAM architecture. We shall see that transactions hold up well when we must add stalling logic to the pipeline.

To start, we need to add the commands `LOAD` and `STORE` to the `Cmd` type:

```
data Cmd = ADD | SUB | INC | LOAD | STORE
```

We also need to define some additional Hawk circuits. The first circuit, `defaultDelay`, augments the normal `delay` circuit so that when a stall hazard is detected, the augmented circuit will output a default value on the next clock cycle, rather than its current input value:

```
defaultDelay :: Signal Bool -> a -> Signal a ->
                Signal a

defaultDelay emitDefault default input =
  delay default (select emitDefault
                        (constant default)
                        input)
```

The `defaultDelay` circuit uses `delay` to store values between clock cycles. The value it stores for the next clock cycle is `default` if `emitDefault` is equal to `True` on the current cycle, otherwise it stores `input`. On the first cycle of the simulation `defaultDelay` always returns `default`.

The `isLoadTrans` circuit returns `True` whenever its argument signal is a load transaction:

```
isLoadTrans :: Signal Transaction -> Signal Bool
isLoadTrans ts = lift isLoad ts
  where
    isLoad (Trans _ cmd _) = (cmd == LOAD)
```

Although we previously passed SHAM instructions as parameters, we now need to call a function, `instrCache`, to explicitly retrieve them:

```
instrCache :: Signal Bool -> Signal Transaction
```

Since the pipeline can stall, we need a way to ask for the same instruction two cycles in a row. The `instrCache` function takes a Boolean signal and returns the current transaction. Whenever the argument signal is `True`, then on the next cycle `instrCache` returns the same transaction as it did for the current clock cycle. Otherwise, it returns the next transaction as normal.

We also need a circuit that actually performs the loads and stores:

```
mem :: Signal Transaction -> Signal Transaction
```

On those clock cycles where the input transaction is anything but a load or store transaction, the `mem` function simply returns the transaction unchanged. On loads, `mem` updates the destination operand of the input transaction, based on the input load address. On stores, `mem` updates its internal memory array according to the address and contents given in the input transaction. The destination operand value is set to zero.

We also define a new Hawk function, **`transHazard`**, that returns `True` whenever its two transaction arguments would cause a hazard, if the first transaction preceded the second transaction in a pipeline:

```
transHazard :: Signal Transaction ->
               Signal Transaction ->
               Signal Bool
```

The extended Hawk specification using transactions is given below:

```
SHAM3Trans :: Signal Transaction
SHAM3Trans = memOut'
  where

     -- register-fetch stage --
     instr = instrCache loadHzd
     readyInstr = regFile memOut' instr
     readyInstr' =
       defaultDelay loadHzd nopTrans readyInstr

     -- ALU stage --
     aluIn = bypass (bypass readyInstr' memOut')
                    aluOut'
     aluOut = alu aluIn
     aluOut' = delay nopTrans aluOut

     -- memory stage --
     memIn = bypass aluOut' memOut'
     memOut = mem memIn
     memOut' = delay nopTrans memOut

  ----- Control logic -----

     loadHzd =
       sigAnd (isLoadTrans readyInstr')
              (transHazard readyInstr'
                           readyInstr)
```

The register-fetch stage retrieves the instruction and fills in its source operands from the register file. The register-fetch pipeline register delays the transaction by one clock cycle, although if there is a load hazard, the register instead outputs a nop-instruction on the next cycle. The ALU stage first updates the source operands of the stored transaction with the results of the two preceding transactions (`memOut'` and `aluOut'`) by invoking bypass twice. It then performs the corresponding ALU operation, if any, on the transaction and stores it in the ALU-stage pipeline register. The memory stage again updates the stored transaction with the immediately preceding transaction, performs any required memory operation, and stores the transaction. The stored transaction is written back to the register file on the next clock cycle. The control logic section determines whether a load hazard exists for the current transaction, that is, whether the immediately preceding transaction was a load instruction that is in hazard with the current transaction.

As we can see, the body of the specification remains manageable. The small control logic section to detect load hazards is straightforward and is a minority of the overall specification. In contrast, an equivalent specification of this pipeline where the components of each transaction were explicitly represented contained over three times as many source lines. The lower-level specification's control section was almost as large as the dataflow section, and not nearly as intuitive.

We feel the transaction ADT is close to the level of abstraction design engineers use informally when reasoning about microprocessor architectures.

4 Modelling the DLX

Using techniques comparable to those described in this report we have modeled several DLX architectures:

- An unpipelined version, where each instruction executes in one cycle.
- A pipelined version where branches cause a one-cycle pipeline stall.
- A more complex pipelined version with branch prediction and speculative execution. Branches are predicted using a one-level branch target buffer. Whenever the guess is correct, the branch instruction incurs no pipeline stalls. If the guess is incorrect, the pipeline stalls for two cycles.
- An out-of-order, superscalar microprocessor with speculative execution. The microarchitecture contains a reorder buffer, register alias table, reservation station, and multiple execution units. Mispredicted branches cause speculated instructions to be aborted, with execution resuming at the correct branch successor.

The microarchitectural specification for the unpipelined DLX is written in a quarter page of uncommented source code; the most complicated pipelined version takes up just over half a page.

4.1 Executing the model

We used the Gnu C compiler that generates DLX assembly to test our specifications on several programs. These test cases include a program that calculates the greatest common divisor of two integers, and a recursive procedure that solves the towers of Hanoi puzzle.

We have not made detailed simulation performance measurements yet. Although we plan to test Hawk on several benchmark programs, we do not expect to break simulation-speed records. Hawk is built on top of a lazy functional language, which imposes some performance costs. Transactions also perform some runtime tests that are "compiled-away" in a lower-level pipeline specification. While it would be nice to get high performance, Hawk is primarily a specification language, and only secondarily a simulation tool. Our main interest is in using Hawk to formally verify microarchitectures, while at the same time retaining the ability to directly execute Hawk programs on concrete test cases.

5 Related Work

There are several research areas that bear a relation on this work, in particular, modeling specific application domains with Haskell, and modeling hardware in various programming languages. We will pick an example or two from these two categories.

Haskell has been used to directly model hardware circuits at the gate level. O'Donnell [10] has developed a Haskell library called Hydra that models gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra has been used to teach advanced undergraduate courses on computer design, where students use Hydra to eventually design and test a simple microprocessor. Hydra is similar to Hawk in many ways, including the use of higher-order functions and lazy lists to model signals. However, Hydra does not allow users to define *composite* signal types, such as signals of integers or signals of transactions. In Hydra, these composite types have to be built up as tuples or lists of Boolean signals. While this limitation does not cause problems in an introductory computer architecture course, composite signal types significantly reduce specification complexity for more realistic microprocessor specifications.

There are many other languages for specifying hardware circuits at varying levels of abstraction. The most widely used such languages are Verilog and VHDL. Both of these languages are well suited for their roles as general-purpose, large-scale hardware design languages with fine-grained control over many circuit properties. Both of these languages are more general than Hawk in that they can model asynchronous as well as synchronous circuits. However, Verilog and VHDL are large languages with complex semantics, which makes circuit verification more difficult. Also, neither of these languages support polymorphic circuits, nor higher-order circuit combinators, as well as Hawk.

The Ruby language, created by Jones and Sheeran [7], is a specification and simulation language based on relations, rather than functions. Ruby is more general than Hawk in that relations can describe more circuits than functions can. On the other hand, existing Ruby simulators require Ruby relations to be *causal*, i.e. to be implementable as functions. Thus Hawk is equal in expressive power to currently executable Ruby programs. In addition, much of Ruby's emphasis is on circuit layout. There are combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on behavioral correctness, so we do not need to address layout issues.

Two other languages that are strongly related are HML [8] and MHDL[2]. HML is a hardware modeling language based on the functional language ML. It also has higher-order functions and polymorphic types, allowing many of the same abstraction techniques that are used in Hawk, with similar safety guarantees. On the other hand, HML is not lazy, so does not easily allow the recursive circuit specifications that turned out to be key in specifying micro-architectures. The goal of HML is also rather different from Hawk, concentrating on circuits that can be immediately realized by translation to VHDL.

MHDL is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL. Though it tackles a very different part of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency and time etc.

6 Future Directions

We have just completed the specification of a superscalar version of DLX, with speculative and out-of-order instruction execution. The use of transactions has scaled well to this architecture; it turns out that superscalar components like *reservation stations* and *reorder buffers* are naturally expressed as queues of transactions.

Beyond this, we intend to push in a number of directions.

- We hope to use Hawk to formally verify the correctness of microprocessors through the mechanical theorem prover Isabelle [11]. Isabelle is well-suited for Hawk; it has built-in support for manipulating higher-order functions and polymorphic types. It also has well-developed rewriting tactics. Thus simplification strategies for functional languages like partial evaluation and deforestation [3] can be directly implemented.

  We also expect that transactions will aid the verification process. Transactions make explicit much of the pipeline state needed to prove correctness. In lower-level specifications this data has to be inferred from the pipeline context.

- We are also working on a visualization tool which will enable the microprocessor engineer to inspect values passing along internal wires.

- We have made initial progress on formally extracting stand-alone control logic from the transaction-based models of pipelines. Stand-alone control logic may be more amenable to conventional synthesis techniques.

7 Acknowledgements

We wish to thank Simon Peyton Jones, Carl Seger, Borislav Agapiev, Dick Kieburtz, and Elias Sinderson for their valuable contributions to this research.

The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews recieves support from a graduate research fellowship with the NSF

References

[1] Aagaard, M., and Leeser, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).

[2] Barton, D. Advanced modeling features of MHDL. In *International Conference on Electronic Hardware Description Languages* (Jan. 1995).

[3] Gill, A., Launchbury, J., and Jones, S. P. A short-cut to deforestation. In *ACM Conference on Functional Programming and Computer Architecture* (Copenhagen, Denmark, June 1993).

[4] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.

[5] Hudak, P., Peterson, J., and Fasel, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.

[6] Johnson, M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[7] Jones, G., and Sheeran, M. Circuit design in Ruby. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.

[8] Li, Y., and Leeser, M. HML: An innovative hardware design language and its translation to VHDL. In *Conference on Hardware Design Languages* (June 1995).

[9] Melham, T. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, G. Birtwhistle and P. A. Subrahmanyam, Eds. Kluwer Academic Publishers, 1988.

[10] O'Donnell, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).

[11] Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[12] Peterson, J., and et al. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.

[13] Sinderson, E., and et al. Hawk: A hardware specification language, version 1. Available at www.cse.ogi.edu/PacSoft/projects/Hawk/, Oct. 1997.

[14] Windley, P., and Coe, M. A correctness model for pipelined microprocessors. In *Second International Conference on Theorem Provers in Circuit Design* (Sept. 1994).

Specifying superscalar microprocessors in Hawk

Byron Cook, John Launchbury, and John Matthews
{byron,jl,johnm}@cse.ogi.edu

Oregon Graduate Institute

Abstract. Hawk is a language for the specification of microprocessors at the microarchitectural level. In this paper we use Hawk to specify a modern microarchitecture based on the Intel P6 with features such as speculation, register renaming, and superscalar out-of-order execution. We show that parametric polymorphism, type-classes, higher-order functions, lazy evaluation, and the state monad are key to Hawk's concision and clarity.

1 Introduction

As the performance of cutting edge microprocessors increases, so too does their microarchitectural complexity. For example:

- A superscalar processor that fetches multiple instructions must cache instructions that cannot be immediately executed.
- A processor with out-of-order execution must usually record the original instruction sequence for exception handling.
- A processor that renames registers must allocate and then recycle virtual register names.

While today's hardware description languages (HDLs) suffice for simple microarchitectures, the features of modern designs are difficult to specify without a richer language. Hawk is a specification language based on Haskell [15] that, for the following reasons, provides a strong foundation for a new generation of HDLs:

- Parametric polymorphism allows generic specifications to be used in different contexts.
- Type-classes provide a convenient mechanism for abstracting over instruction sets, register sets, and microarchitectural components.
- Higher-order functions enable a designer to structure specifications in elegant and concise ways.
- Lazy evaluation naturally supports the simulation of multiple mutually dependent streams of instructions and data.
- The state monad facilitates a disciplined style when specifying components with mutable state.

In this paper we explore a Hawk specification of a microarchitecture based on the Intel P6 [4]. We give an overview of the top-level design, and describe in detail our specification of the Reorder Buffer. The purpose of this paper is to show that complex microarchitectures can be formally specified in a clear, concise and intelligible way that facilitates understanding, design review, simulation, and verification.

We assume the reader is familiar with the basic concepts of functional languages and microarchitectural design (such as branch prediction and pipelining). For an in-depth introduction to Haskell, read Hudak, Peterson, and Fasel's tutorial [5]. For more information on microarchitectures, refer to Johnson's textbook [6].

The remainder of this paper is organized as follows: in Section 2 we introduce an architecture; in Section 3, we provide an introduction to Hawk; in Section 4 we use Hawk to specify the architecture; and in Section 5 we highlight how the features of Hawk are used in the specification.

2 A modern microarchitecture

2.1 Machine instruction notation

Throughout this paper we use the following notation for machine instructions:

```
r1 <- r2 + r3
```

The register `r1` is the *destination register* or *destination operand.* Registers `r2` and `r3` are *source registers.*

When the contents of a register is known we may choose to pair the register name and its value:

```
r1 <- (r2,5) + r3
```

In this case, `5` is a *source register value.*

When an instruction's *destination register value* has been computed, we denote this by pairing the destination register with its value:

```
(r1,8) <- (r2,5) + (r3,3)
```

We sometimes refer to a destination register value as the instruction's value.

2.2 Superscalar microarchitectures

In general, superscalar architectures employ aggressive strategies to resolve inter-instruction dependencies and mask the latency of memory accesses. These include speculative execution, the use of virtual register names, and out-of-order instruction issue. The internal microarchitectures often resemble that of a dataflow processor using speculative parallel evaluation. They are thus able to exploit instruction level parallelism to execute sequential, scalar programs.

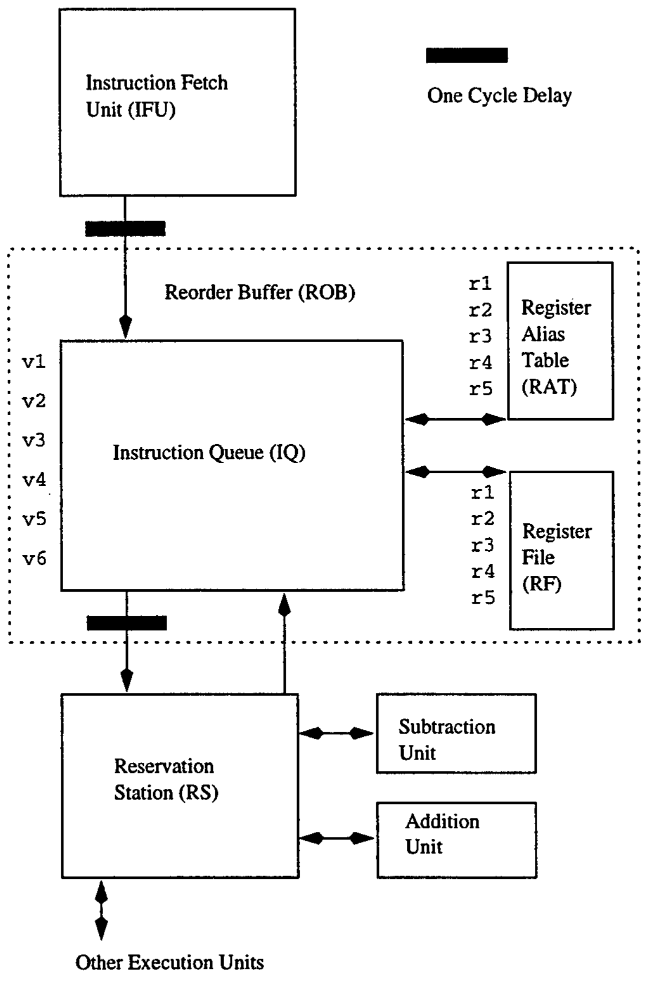


Fig. 1. Microarchitecture

The focus of this paper is on the speculative, superscalar, out-of-order, register renaming microarchitecture shown in Fig. 1. In the remainder of this section we provide an informal introduction to the architecture.

A Reorder Buffer (ROB) maintains the sequential programming model of an architecture while instructions are executed out-of-order and in parallel elsewhere in the processor. In Fig. 1 the ROB is shown as the composite of a circular Instruction Queue, a Register Alias Table, and a Register File for the real register set.

The Instruction Queue (IQ) stores instructions in the order in which they are received from the Instruction Fetch Unit (IFU). The IQ also behaves like a register file for the virtual register set, where the instruction's position in the IQ is its virtual register name.

The Register Alias Table (RAT) is an array of virtual register names indexed by the real register set. For a given real register name, *r*, the RAT contains either the location of the youngest instruction in the IQ using *r* as a destination operand; or nothing, if no instruction in the ROB contains the destination operand *r*. For example, if the instruction `r5 <- r2 + r3` is placed into position `v1` of the IQ (as in Fig. 2), then the real register `r5` is aliased in the RAT to the virtual register `v1`. If `r4 <- r5 + r2` is then inserted into the IQ (Fig. 3), its reference to `r5` is updated to `v1`, and `r4` is aliased to `v2` in the RAT.

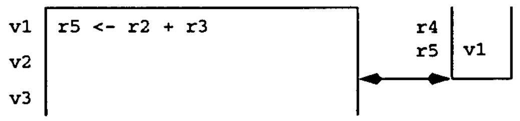


Fig. 2. Inserting `r5 <- r2 + r3` into the ROB

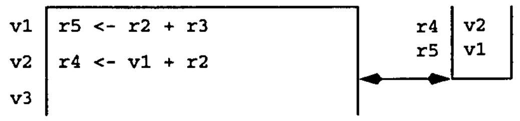


Fig. 3. Inserting `r4 <- r5 + r2` into the ROB

Each instruction, after it has been placed into the ROB, is passed onto the Reservation Station (RS) to be executed. The RS is a data-flow circuit that can execute instructions out-of-order and in parallel. Upon completion in the RS, an instruction's value is returned to the ROB and forwarded to other instructions still in the RS.

2.3 Retiring instructions

An instruction is retired from the ROB when it is at the front of the IQ and its value has been calculated. To retire an instruction in location v with destination operand r, the ROB must write the instruction's value to position r in the Register File, and remove the alias from the RAT if r is still aliased to v.

Why isn't r always aliased to v? Consider the scenario in Fig. 4, where the ROB contains two instructions with `r5` as their destination operand. The virtual register `v1` is no longer an alias of `r5` in the RAT. When retiring the instruction from `v1`, the alias in the `r5` position of the RAT should not be removed. Doing so would remove the unrelated alias from `r5` to `v3`. However, in Fig. 5, because only

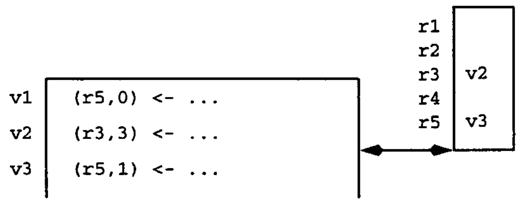


Fig. 4. IQ contains two instructions that alter `r5`

one instruction contains the destination operand `r5`, `r5` remains aliased to `v1`. In this case, when retiring instruction `v1` from the IQ, the alias at the position `r5` in the RAT should be erased.

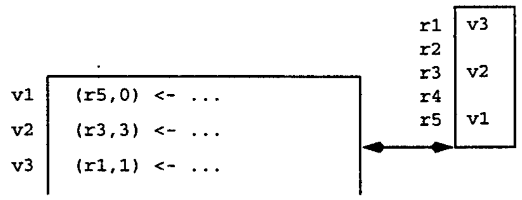


Fig. 5. IQ contains one instruction that alters `r5`

2.4 Example

To illustrate the microarchitecture in action, we trace the execution of a four instruction program:

```
r2 <- r1 + r3
```

```
r4 <- r4 + r2
r2 <- r1 + r1
r1 <- r5 - r3
```

Rather than demonstrating the potential performance of the microarchitecture, this example is tailored to show the amount of bookkeeping that the processor must maintain.

In Fig. 6, execution begins in Cycle 1 with the fetch of four instructions, the last of which requires a different execution unit. In Cycle 2 the fetched instructions are inserted into the IQ. Source register references are modified in one of two ways. Either the operand is replaced with a virtual register reference if it is aliased in the RAT, or the register's value is filled in from the Register File. During Cycle 3 the first and last instructions are executed in parallel. In Cycle 4 the ROB begins retiring instructions based on their position in the instruction sequence. Although the first and last instructions have both completed, to maintain the sequential programming model, only the first instruction can be retired. The last instruction remains in the ROB until its predecessors have all been retired. In Cycle 5, `v2` is computable because the value of `v1` has been forwarded to the source operand. In Cycle 6, because instruction `v2` has completed, the remaining instructions are retired.

3 The Hawk specification language

This section introduces concepts and abstractions used in Hawk. At the risk of incompleteness, we will rely on the reader's intuition to fill in the meanings of functions and syntax that are not described.

3.1 Signals

A signal represents a wire, where at each clock cycle the value of a signal may change. For example, a signal could alternate between `True` and `False`. Or a signal might contain a series of primes numbers. Informally, we can think of a signal as an infinite sequence where the clock cycle is the index:

```
toggle = True, False, True, False, True, False, ....
primes = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,  ....
```

Like the synchronous language Lustre [3], Hawk provides a built-in signal type and functions to construct and manipulate them. The function `constant`, from Fig. 7, returns a signal that does not change over time:

```
constant 5 = 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ....
```

The function `before` delays a signal with a list of initial values[1]:

```
[-1,0] ‘before‘ primes = -1, 0, 2, 3, 5, 7, 11, ....
```

[1] `‘before‘` denotes that `before` is used as an infix operator

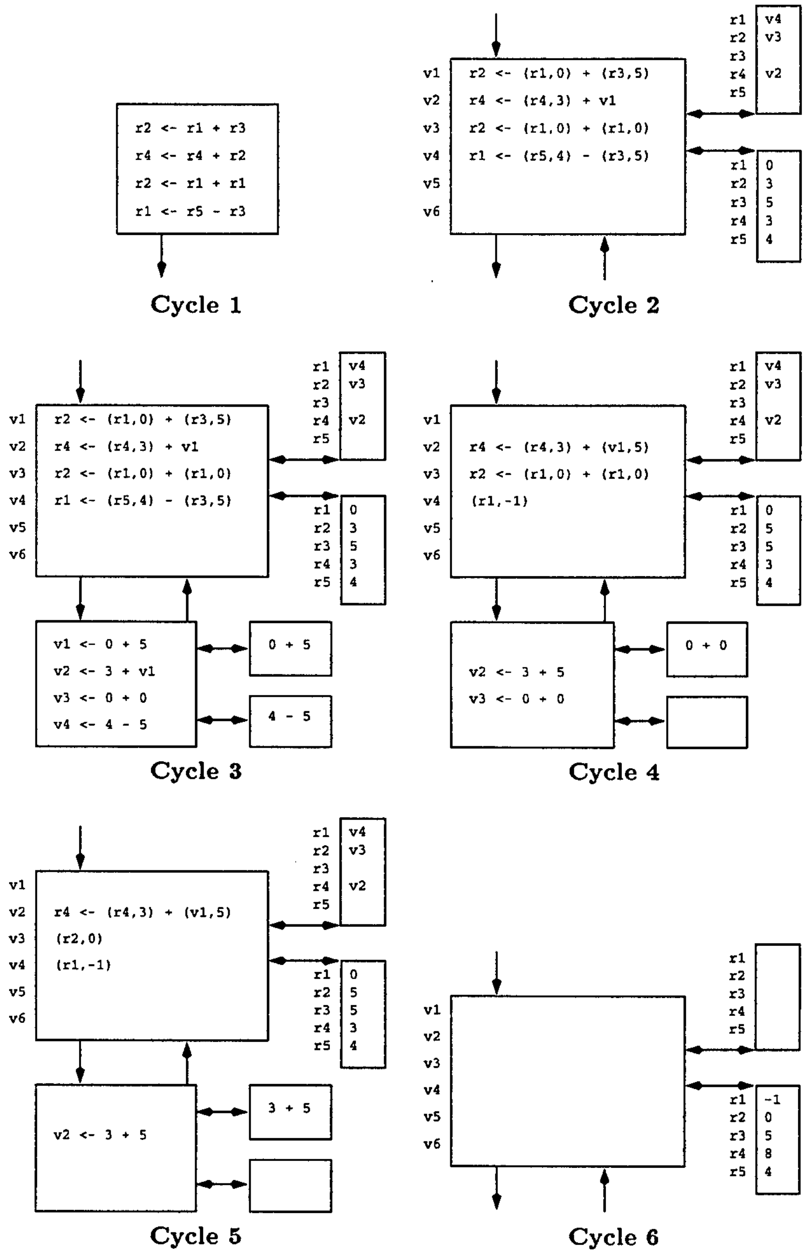


Fig. 6. Example execution trace

```
constant :: a -> Signal a
delay    :: a -> Signal a -> Signal a
before   :: [a] -> Signal a -> Signal a
bundle   :: (Signal a,Signal b) -> Signal (a,b)
unbundle :: Signal (a,b) -> (Signal a, Signal b)
lift     :: (a -> b) -> Signal a -> Signal b
```

Fig. 7. Type signature of primitive `Signal` functions

The function `bundle` takes a pair of signals and returns a signal of pairs:

```
bundle (primes,toggle) = (2,True), (3,False), ....
```

The function `lift` applies its argument to each value in a signal:

```
lift f primes = f 2, f 3, f 5, f 7, f 11, ....
```

Conditional statements are overloaded for signaled expressions. For example:

```
if toggle then primes
          else constant 0    = 2, 0, 5, 0, 11, ....
```

Later in this paper we use the function `delay`, which is defined in terms of `before`:

```
delay x s = [x] 'before' s
```

So, for example:

```
delay 6 primes = 6, 2, 3, 5, 7, 11, 13, 17, ....
```

3.2 Transactions

Transactions [1] formalize the notation of instructions introduced in Subsection 2.1. A transaction is a machine instruction grouped together with its state. This state might include:

- Operand values.
- A flag indicating that the instruction has caused an exception.
- A predicted jump target, if the instruction is a branch.
- Other obscure information, such as predicted operand values if we choose to implement value locality [12] optimizations.

Transactions are provided as a library of functions, written in Hawk, for creating and modifying transactions. For example, `bypass` takes two transactions and builds a new transaction where the values from the destination operands of the first transaction are forwarded to the source operands of the second. If `i` is the transaction:

```
(r4,8) <- (r2,4) + (r1,4)
```

and j is the transaction:

```
r10 <- (r4,6) + (r1,4)
```

then bypass i j produces the transaction:

```
r10 <- (r4,8) + (r1,4)
```

In our experience, specifications that operate on transactions are more concise than those that treat instructions and state separately. When designed in this style, a processor fetches a transaction containing only the machine instruction which is later refined by the various microarchitectural components until the destination operand value is calculated. Transactions are an example of a user-defined abstraction designed to aid the development of a complex microarchitecture. The concept of an instruction's local state as it acquires its operands, is executed, and finally retired, is the essential concept of a superscalar processor. Transactions also aid the verification process because they make explicit much of the state needed to prove correctness. In lower-level specifications this data has to be inferred from the context.

4 Specifying the microarchitecture

Fig. 8 contains the top-level Hawk specification of the microarchitecture in Fig. 1. Using lazy evaluation, a Hawk simulation will solve the specification's system of mutually dependent equations, producing a computational simulation. The components of the microprocessor are modeled as functions from input signals to output signals. For example, as Fig. 9 illustrates, the ROB is a component with two inputs and four outputs. The inputs and outputs may each represent very wide connections — perhaps enough to move numerous transactions in a single cycle. The arguments and results of the function rob from Fig. 8,

```
(retired,ready,n,err) = rob 6 (fetched,computed)
```

except for the size parameter, correspond to those in Fig. 9.

4.1 Top-level structural specification

In Fig. 8 the first equation specifies how transactions are fetched from the instruction memory, mem:

```
(instrs,npc) = ifu 5 mem pc err ([5,5] `before` n)
```

The Instruction Fetch (IFU) function, ifu, uses its first parameter, 5, to determine the maximum number of transactions to fetch at each cycle. The IFU retrieves consecutive transactions beginning at the program counter, pc. Initially, during the first and second cycles, 5 transactions are fetched. In later cycles feedback from the ROB, n, is used to determine the number of transactions to fetch.

Execution begins with the transaction at location 256 in the instruction memory. After the first cycle, the value of pc depends on the location of the

```
processor mem = retired
 where

 (instrs,npc) = ifu 5 mem pc err ([5,5] 'before' n)

 pc = delay 256 (if err then lastpc retired else npc)

 fetched = delay [] (annotate instrs)

 (retired,ready,n,err) = rob 6 (fetched, computed)

 computed = rs (6,execUnits) (delay False err,delay [] ready)

 memU = mob fetched retired

 execUnits = [addU,subU,jmpU,intU,fltU,memU]
```

Fig. 8. Top-level microprocessor specification

previously fetched transaction, and the possibility of a mispredicted branch or exception. In the event of a mispredicted branch or exception, the signal `err` is set, and the pc comes from the last retired transaction:

```
pc = delay 256 (if err then lastpc retired else npc)
```

For simplicity we employ a naive branch prediction algorithm — all branch transactions are simply assumed to jump to the next consecutive transaction. The function `annotate` places this guess into the state of branch transactions:

```
fetched = delay [] (annotate instrs)
```

The Reservation Station (RS) function, `rs`, is parameterized on its size and execution units:

```
computed = rs (6,execUnits) (delay False err,delay [] ready)
```

During the initialization of `rs`, the execution units are clustered together with a function. The execution units can be pipelined or blocking. Execution units can also complete in multiple clocks. The RS accepts two input signals: an error flag and transactions from the ROB. The transactions `computed` contains the transactions that are complete and ready to be updated in the ROB.

4.2 ROB specification

Whereas the top-level specification of the microarchitecture is easily constructed as a purely functional application of components, the ROB is more complicated. Certainly the ROB could be specified in the applicative style used in Fig. 8. However, at a higher level of abstraction, the ROB can be thought of as a circuit

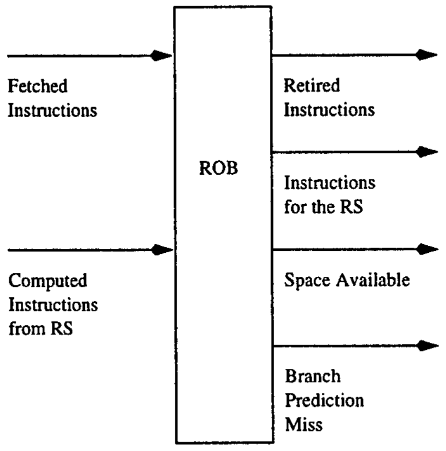


Fig. 9. Inputs and outputs of the ROB

that sequences destructive updates on mutable components. Our approach in this paper is to specify the ROB in a behavioral style using imperative language features. In Fig. 10, the specification of the ROB is provided in the state monad and then encapsulated with Hawk's state thread encapsulation construct `runST` [9]. The advantage of using `runST` is that the language guarantees that `rob` neither depends on nor alters mutable state in other components or an outside environment [10]. We can therefore treat the ROB as a pure function that, on a given input, always returns the same output.

In Fig. 10, during the beginning of the simulation, the ROB constructs its mutable sub-components (much of this work would be fabricated into the processor):

```
q <- IQ.new n
rat <- RAT.new
rf <- RF.new
```

At each cycle the ROB takes the `fetched` and `computed` signals signals

```
cycle(fetched,computed)
```

and performs the following tasks:

- Update the `computed` transactions in the queue. For each transaction in the `computed` list, the function `update` obtains the virtual register reference from the destination register, and uses it as the index when updating the queue:

  ```
  update q computed
  ```

- Insert the `fetched` transactions into the queue (see Subsection 4.3):

  ```
  instrs <- insert rat q rf fetched
  ```

- Find transactions from the front of the queue that are ready to be retired. If a retired transaction was a mispredicted branch or raised an exception, then only retire the transactions before it (see Subsection 4.4):

```
(retired,err) <- retire rat q rf
```

- If a retired transaction was a mispredicted branch or raised an exception, then clear the IQ and RAT:

```
if err then do {q.clear; rat.clear}
```

- Measure the capacity of the queue for the IFU:

```
capacity <- q.space
```

- If a retired transaction was mispredicted or raised an exception then do not send `fetched` transactions to the RS:

```
let ready = if err then [] else instrs
```

- Return the retired transactions, the transactions ready to pass onto the RS, the measured capacity, and the error flag:

```
return (retired,ready,capacity,err)
```

```
rob n (fetched,computed)
  = runST (
    do { q <- IQ.new n
       ; rat <- RAT.new
       ; rf <- RF.new
       ; cycle(fetched,computed)
            { update q computed
            ; instrs <- insert rat q rf fetched
            ; (retired,err) <- retire rat q rf
            ; if err then do {q.clear; rat.clear}
            ; capacity <- q.space
            ; let ready = if err then [] else instrs
            ; return (retired,ready,capacity,err)
            }
       }
    )
```

Fig. 10. ROB behavioral specification

```
insert rat q rf instrs
  = foreach t in instrs
    do { (reg,alias) <- q.assignAddr (head (getDestRegs t))
       ; src <- mapM (rat.replace) (getSource t)
       ; rat.write reg alias
       ; dest <- mapM (rat.replace) (getDest t)
       ; new <- regRead q rf (trans dest (getOp t) src)
       ; q.enQueue new
       ; return new
       }
```

Fig. 11. Insertion specification

4.3 Inserting new instructions

Fig. 11 contains the specification of the function `insert`. When inserting new transactions into the ROB, `insert` takes a list of transactions, `instrs`, and performs the following actions:

- Calculate the new position in the queue for the transaction:

  ```
  (reg,alias) <- q.assignAddr (head (getDestRegs t))
  ```

- Substitute references to real registers with virtual registers in the source operands:

  ```
  src <- mapM (rat.replace) (getSource t)
  ```

- Update the RAT:

  ```
  rat.write reg alias
  ```

- Substitute the reference from the real destination register to the virtual destination register:

  ```
  dest <- mapM (rat.replace) (getDest t)
  ```

- Read real register references:

  ```
  new <- regRead q rf (trans dest op src)
  ```

- Enqueue the transactions:

  ```
  q.enQueue new
  ```

- Return the updated transactions:

  ```
  return new
  ```

```
retire rat q rf
  = do { perhaps <- q.deQueueWhile complete
       ; let (retired,err) = hazard findErr perhaps
       ; mapM (writeOut rf rat) retired
       ; return (retired,err)
       }
  where findErr t = jmpMiss o exceptionRaised

jmpMiss t = do { x <- getPC t
               ; y <- getSpecPC t
               ; return (x /= y)
               }
            `catchEx` False

writeOut rf rat t =
 do { let [Reg (Virtual vr real) (Val x)] = getDest t
    ; rf.write real x
    ; a <- rat.read real
    ; do { v <- a ; guard (v == vr) ; return (rat.remove real) }
      `catchEx` return ()
    }
```

Fig. 12. Retirement specification

4.4 Retiring instructions

Fig. 12 contains the specification of the function `retire`. When retiring transactions from the ROB, `retire` performs the following actions:

- Remove transactions from the front of the queue until a transaction is found that has not been computed:

  ```
  perhaps <- q.deQueueWhile complete
  ```

- If a branch was mispredicted or an exception was raised then ignore all of the transactions after that transaction:

  ```
  let (retired,err) = hazard findErr perhaps
  ```

- Write the values of the destination registers to the Register File :

  ```
  mapM (writeOut rf rat) retired
  ```

- Return the retired transactions and a flag indicating a branch miss or raised exception:

  ```
  return (retired,err)
  ```

5 Conclusions

The design of correct superscalar microarchitectures is difficult. The language of discourse must be powerful enough to describe a wide range of processors, and concise enough that designers can maintain intellectual control of their work. Moreover, the language must scale to the designs of the future. In this section we highlight how polymorphism, type-classes, higher-order functions, lazy evaluation and the state monad improve the concision, clarity, and perhaps the provability of our specification.

5.1 Polymorphism

Many of Hawk's library functions are polymorphic. For example, `delay` accepts an argument of type `a` (where `a` could be any type), a signal of `a`, and returns a new signal of `a`:

```
delay :: a -> Signal a -> Signal a
```

In Fig. 8, `delay` is used on both Booleans and lists of transactions:

```
(delay False error, delay [] ready)
```

Without parametric polymorphism, a delay function would be required for each specific type. In many specification languages, because the types that can be passed through signals are limited, ad hoc solutions are usually sufficient. However, signals in Hawk are unrestricted and therefore must be accompanied by truly polymorphic functions.

5.2 Type-classes

The RAT, used in Fig. 10, is abstracted over the register set used in the underlying machine language. For example, the function `RAT.new` is of type:

```
RAT.new :: Register r => ST s (RAT s r v)
```

This reads "for any type `r` that is a register set, `RAT.new` constructs a new RAT indexable by `r`". Because `r` is an instance of `Register`, the variables `minBound` and `maxBound` are overloaded to the smallest and largest values of `r`:

```
minBound :: Register r => r
maxBound :: Register r => r
```

`RAT.new` uses `minBound` and `maxBound` to determine the size of the constructed RAT.

Without type-classes, the RAT would either be useful for only one particular register type, or a number of extra parameters (such as the bounds and comparison functions) would need to be passed to the functions `rob`, `RAT.new`, `insert`, etc. Type-classes allow us to easily adapt the RAT to other machine languages, such as IA-64 or PowerPC.

5.3 Higher-order functions

Higher-order functions allow designs to be parameterized in new and powerful ways. For example, in Fig. 8 the RS is parameterized over a list of execution units. At the start of a simulation, the RS builds a single execution unit by clustering the list of circuits. When testing various microarchitectural configurations, the designer can easily modify the execution units at the top-level.

We might also want to abstract the RS on the scheduling function:

```
computed = rs (6,cluster,[addU,subU,jmpU,mltU])
              (delay False error,delay [] ready)
```

This way we might use the same RS specification in many instantiations with different configurations of scheduling functions and execution units.

5.4 Lazy evaluation

Without Hawk's lazy semantics we would not be able to write the dependent equations in Fig. 8. Consider the simple clock circuit in Fig. 13. The design is

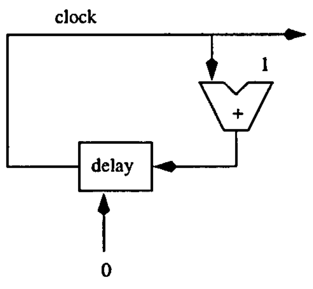


Fig. 13. Clock circuit

easily specified as a Hawk expression where the value depends on itself:

```
clock = delay 0 (clock + 1)
```

In a strict semantics, the meaning of this expression would be undefined.

5.5 Encapsulated state

While maintaining the mathematically consistent features of Hawk, such as polymorphism and lazy evaluation, the state monad adds the ability to use mutable state directly rather than encoding state with delays and other lower level signal constructs. The use of runST facilitates the safe integration of imperative specifications in an applicative framework.

6 Future work

Currently, using the Glasgow Haskell Compiler, the simulator derived from the specification in this paper retires 800 instructions per second when executed on a UltraSPARC-1 processor. We hope that to improve performance using domain specific optimizations or compilation to better simulation packages.

We have not sufficiently explored the synthesis and analysis of Hawk specifications. Although Hawk is at a higher level of abstraction than mainstream HDLs from our initial results we believe that, within limits, automatic synthesis is feasible.

We have just completed a correctness proof of a microarchitecture based on this paper in which the ROB, RS, and IFU are specified axiomatically [8]. We now hope to prove that the ROB, RS, and IFU presented here implement the axioms.

We hope to use Hawk formally to verify the correctness of microprocessors with a mechanical theorem prover (for example, Isabelle [14]). A theorem proving environment for Hawk must have support for manipulating higher-order functions and polymorphic types.

7 Related work

Ruby [7] is a specification language based on relations, rather than functions. Relations can describe more circuits than functions. Much of Ruby's emphasis is on circuit layout. Ruby provides combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on circuit correctness, so we do not address layout issues.

Lava is a Haskell library for the specification of Field Programmable Gate Arrays. Lava is intended to be used at a lower level of abstraction than Hawk. Like Ruby, Lava specifications focus much attention on issues related to layout.

Like Hawk, Lustre [3] and the other reactive synchronous languages (Signal, Esterel, Argos, etc) provide mechanisms for constructing expressions over time-varying domains. However, research on these languages has emphasised reactive features rather than the issues addressed in this paper.

The Haskell library Hydra [13] allows modeling of gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra is similar to Hawk in many respects. However composite signal types, such as signals of integers, must be constructed as tuples or lists of Boolean signals. This restriction severely limits Hydra's application to the domain of complex microarchitectures.

HML [11] is a hardware modelling language based on ML. It supports higher-order functions and polymorphic types, allowing many of the same abstraction techniques that are used in Hawk. On the other hand, HML is not lazy, so it does not easily allow the dependent circuit specifications that are key in specifying microarchitectures in Hawk. Also, HML does not clearly separate its imperative and functional features.

MHDL [2] is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL. Though it tackles a very different area of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency, time, etc.

8 Acknowledgements

For their contributions to this research, we thank Borislav Agapiev, Mark Aagaard, John O'Leary, Robert Jones, Todd Austin, and Carl Seger of Intel Corporation; Elias Sinderson and Neil Nelson of The Evergreen State College; Dick Kieburtz, Jeff Lewis, Sava Krstić, Walid Taha, and Andrew Tolmach of the Oregon Graduate Institute; Simon Peyton-Jones of the University of Glasgow; and the members of BWERT.

This research was supported by Intel and Air Force Material Command (F19628-93-C-0069). John Matthews is supported by a fellowship from the National Science Foundation.

Note: This paper appears in the proceedings of the 1998 Workshop on Formal Techniques for Hardware (Marstrand, Sweden)

References

1. Aagaard, M., and Leeser, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).
2. Barton, D. Advanced modeling features of MHDL. In *International Conference on Electronic Hardware Description Languages* (Jan. 1995).
3. Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages* (Munich, Germany, Jan. 1987).
4. Gwennap, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995).
5. Hudak, P., Peterson, J., and Fasel, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.
6. Johnson, M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.
7. Jones, G., and Sheeran, M. Circuit design in Ruby. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.
8. Krstić, S., Cook, B., Launchbury, J., and Matthews, J. A correctness proof of a speculative, superscalar, out-of-order, renaming micro-architecture. Submitted to 1998 Formal Methods in Computer Aided Design, Apr. 1998.
9. Launchbury, J., and Jones, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.
10. Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In *International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

11. LI, Y., AND LEESER, M. HML: An innovative hardware design language and its translation to VHDL. In *Conference on Hardware Design Languages* (June 1995).
12. LIPASTI, M. H. *Value Locality and Speculative Execution.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1997.
13. O'DONNELL, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).
14. PAULSON, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
15. PETERSON, J., AND ET AL. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.

A correctness proof of a speculative, superscalar, out-of-order, renaming microarchitecture — extended technical report

Sava Krstić, Byron Cook, John Launchbury, and John Matthews
{krstic,byron,jl,johnm}@cse.ogi.edu

Oregon Graduate Institute

Abstract

Microarchitects are increasingly using techniques such as speculation, register renaming, and superscalar out-of-order execution to make use of instruction-level parallelism. However, the growing complexity of modern microprocessors exacerbates the difficulty of relating them to the simple machines that they emulate. Flaws found later in lower-level validation are often microarchitectural in nature.

In this paper we provide high-level mathematical specifications for a basic machine and for a speculative, superscalar, out-of-order, renaming machine based on the Intel P6 microarchitecture. We then prove that the visible outputs of the two machines are equivalent.

1 Introduction

As the performance of microprocessors increases, so too does their microarchitectural complexity. Modern architectures employ aggressive strategies to resolve inter-instruction dependencies. These include rich combinations of speculative, superscalar, and out-of-order execution with the use of virtual register names. Proving that a microarchitecture with these features implements the architecture's programming model is extremely difficult. However, it is an important aspect of design because flaws found later during lower-level validation are frequently manifestations of errors in the microarchitectural specification.

The limited real use of verification in practice is primarily attributable to the immaturity of the techniques, rather than a lack of desire. Industry is working hard to find formal verification methods that scale to the problem sizes they face. Our paper attempts to address some aspects of this issue

Our research is based on a fairly detailed model of a P6-like microarchitecture [4, 6] expressed using Hawk [7]. To prove its correctness, we have constructed a more abstract specification in which each major component is axiomatically specified. We are then able to prove that, for any given program, the *visible output* computed by the microarchitecture is identical to that of the simple reference machine.

The model described in this paper implements speculation, prediction, superscalar out-of-order execution, and renaming. Following Intel convention, we will refer to this combination of microarchitectural optimizations as *dynamic execution.* Actually, our axiomatization does not limit us to a single microarchitecture. The proof is applicable to any combination of components that satisfy the axioms, so our result should be relatively easily adaptable to other architectures that use elements of dynamic execution, such as the MIPS T5, HP's PA8000, Digital's Alpha, etc.

We believe this approach to demonstrating correctness would be feasible for use in industry. Of course, even the moderately complex model we have here is several orders of magnitude simpler than a commercial design, but the hierarchical nature of the proof is promising. As each design team is developing an RTL description of a component, the particular axioms make explicit the assumptions that other teams can rely on. If these axioms have to change during development then there is opportunity to determine that the global correctness property still holds, and if not, what explicit changes need to be made to other units.

This paper is organized as follows: we specify a simple machine, provide a specification of the dynamic machine, and prove that the microarchitectures are visibly equivalent.

2 Defining correctness: the standard machine

A microprocessor's correctness is typically defined by the instruction set architecture (ISA), which gives semantics to each instruction in terms of the machine's states (register files, caches, etc). We adopt a slightly different perspective, abstracting away the ISA in the concept of a simple *standard machine.* This decision stems from the fact that top-level specifications of dynamic architectures are largely independent of the concrete ISA. Instead, they are distinguished by the way they treat dependencies and branching in programs, and those essentials are captured in our standard machine.

Concrete ISAs should be thought of as refinements of the standard machine, and for each such refinement there is a corresponding refinement of the dynamic machine described in the next section. This makes our correctness proof, with some extra work to define refinements, valid for a wide class of ISAs.

We assume that the standard machine executes a fixed program, so its result can simply be described as a sequence of pairs of the form (instruction, result). Two sorts are needed for this: `PgmIdx` for indices (addresses) of instructions, and `Value` for results and operands. We make three assumptions about the standard machine:

- It executes a sequence of instructions, where each instruction in the sequence is determined by the preceding instruction and its result.
- The result of any instruction can be computed if the values of the two operands are known.
- The operands of each instruction are results of some previously executed instructions, or a default value.

Three functions suffice to model this situation:

$$\begin{aligned}\texttt{compute}&: \texttt{PgmIdx}, \texttt{Value}, \texttt{Value} \to \texttt{Value},\\ \texttt{next}&: \texttt{PgmIdx}, \texttt{Value} \to \texttt{PgmIdx},\\ \texttt{getSources}&: \texttt{PgmIdx}, \texttt{PgmIdxSeq} \to \mathbf{N}+(), \mathbf{N}+(),\end{aligned}$$

where $\mathbf{N}+()$ is the set of natural numbers plus a distinguished value $()$.

The result of the execution of the standard machine, or the *visible output*, is an infinite sequence

$$\texttt{standard} = (i_0, v_0), (i_1, v_1), \ldots$$

defined inductively by the following axioms.

Axiom SM-1. $i_0 = \texttt{startIdx}$.

Axiom SM-2. $i_m = \texttt{next}(i_{m-1}, v_{m-1})$, for $m > 0$.

Axiom SM-3. $v_m = \texttt{compute}(i_m, (v_p, v_q))$,
where $(p, q) = \texttt{getSources}(i_m, (i_1, \ldots, i_{m-1}))$, and $v_{()} = \texttt{defaultValue}$.

Axiom GS. If $\texttt{getSources}(i, (i_1, \ldots, i_m)) = (p, q)$, then each of p, q is either $()$ or an integer smaller than m.

These axioms assume two constants, the starting instruction $\texttt{startIdx} \in \texttt{PgmIdx}$ and $\texttt{defaultValue} \in \texttt{Value}$.

We think of the standard machine as executing one instruction per cycle. Axiom SM-2 states that the function `next` determines the location of the program counter at cycle m based on the instruction and value from $m-1$. So, for example, $\texttt{next}(i, v) = i+1$ for all but branch instructions. Axiom SM-3 states that `getSources` returns the cycles (p and q) at which the source operands for instruction i_m are calculated. The value v_m is then defined in terms of the values at cycles p and q.

The virtue of the standard machine is in its unified treatment of instructions and its use of a small number of functions capable of expressing the inter-instruction relationships which are at the basis of more sophisticated execution algorithms. With this simplicity there comes an important limitation: the standard machine does not have enough specification details to properly model dependencies between instructions that manipulate the memory. These dependencies are established only after address computation, and our `getSources` is "static" in that respect.

The standard machine models register dependencies fully. As regards the memory dependencies, it only supports the basic model in which these instructions are done in-order. This is achieved by defining `getSources` so that each load or store instruction is dependent on the last preceding store.

3 Specifying the dynamic microarchitecture

In this section we introduce the specification of the dynamic microarchitecture in Figure 1, which is based on the Intel P6 microarchitecture. It is composed of the following components:

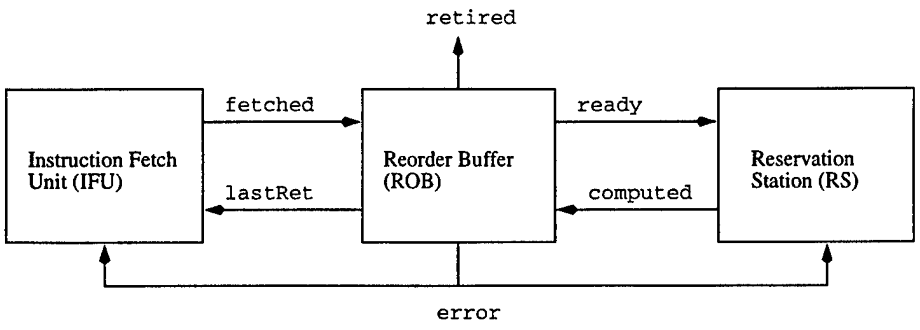


Figure 1: Top-level dynamic microarchitecture

Instruction Fetch Unit (IFU). The IFU provides multiple instructions at each clock cycle and sends them to the ROB through the `fetched` wire. The IFU also adds information to the fetched instructions. For example, since the IFU would typically use branch prediction, each branch instruction in `fetched` is annotated with its speculative program counter.

Reorder Buffer (ROB). The ROB maintains the sequential programming model of the ISA while instructions are executed in parallel elsewhere in the processor. In essence the ROB is a queue of instructions. After enqueing instructions from the `fetched` wire, the ROB passes them on to the Reservation Station through the `ready` wire. An instruction can be dequeued when its value has been computed in the Reservation Station and all of the instructions that were fetched before it have been dequeued. In the case of a mispredicted branch, the ROB asserts the `error` signal and returns the program counter to the IFU through the `lastRet` signal. The visible output of the microarchitecture is the `retired` wire, which represents the instructions retired at each clock cycle.

Reservation Station (RS). The RS is the data-flow execution component of the microarchitecture. Instructions placed into the ROB are passed on to the RS through the `ready` wire. The RS can execute instructions dynamically. Upon completion in the RS, an instruction's value is forwarded to other instructions still in the RS, and eventually returned to the ROB through the `computed` wire.

3.1 Concepts used in the formal specification

3.1.1 Transactions

We think of instructions in the execution process as entities which come into being at a certain cycle and evolve thereafter. To formalize those entities, we use the concept of *transactions* [1, 7]. A transaction is a package of information which (directly or indirectly) contains the identity of the unique instruction it is associated with plus various data contained in the current machine state that are relevant for the execution of that instruction. Pairs (i, v) in the description of the standard machine are a simple example of transactions. In general, the structure of transactions depends on the machine being considered and is a matter of choice. Here we use six components:

$$\texttt{Trans} = \texttt{PgmIdx}, \texttt{PgmIdx}, \texttt{Name}, \texttt{Ops}, \texttt{Result}, \texttt{Status}.$$

The first component is the index of the instruction in the (fixed) program and the second is the index of the speculative next instruction. The Name component provides unique identifiers to instructions; we take Name to be the set of positive integers, so that the name of an instruction will be its index in the sequence of all fetched instructions. Next we have $\texttt{Result} = \texttt{Value} + \texttt{Name}$ and $\texttt{Ops} = \texttt{Value} + \texttt{Name}, \texttt{Value} + \texttt{Name}$. Thus, all instructions have two operands, and the sort Value is conveniently extended with Name to include references to values that are not yet computed. This is the essence of register renaming. Finally, Status is the finite set of letters $\{A, C, D, E, N, R\}$, abbreviating the words Active, Computed, Dropped, Error, New, and Retired respectively.

The projections from Trans to its six components will be denoted **pc**, **spc**, **name**, **ops**, **res**, and **sts** respectively. Angle brackets will be used for projections onto several components; for example, $\langle \mathbf{pc}, \mathbf{res} \rangle \colon \texttt{Trans} \to \texttt{PgmIdx}, \texttt{Result}$. The two operands will be denoted by **op1** and **op2**; thus, $\mathbf{ops}(t) = (\mathbf{op1}(t), \mathbf{op2}(t))$.

3.1.2 Signals

Another important concept in the specification is that of *signals*. A signal represents a wire, where at each clock cycle the value may change. We think of signals as infinite sequences indexed by the clock cycles. If s is a signal, then s_n denotes its n^{th} element, i.e., the value of s at clock cycle n.

Particularly convenient in high-level specifications are signals of transactions. Even though a physical wire would never contain a whole transaction, its content is usually associated with a unique transaction. Refinements to lower level specifications could replace transaction signals with the relevant data components.

3.2 Top-level specification and correctness statement

Recall that the dynamic machine is composed of an IFU, ROB, and RS (Figure 1). The top-level specification of the dynamic machine is given by mutually recursive equations

$$\begin{aligned} \texttt{fetched} &= \mathit{ifu}(\texttt{lastRet}, \texttt{error}) \\ (\texttt{retired}, \texttt{ready}, \texttt{lastRet}, \texttt{error}) &= \mathit{rob}(\texttt{fetched}, \texttt{computed}) \\ \texttt{computed} &= \mathit{rs}(\texttt{ready}, \texttt{error}) \end{aligned}$$

which define the signals fetched, computed, ready, retired, error, and lastRet. The functions *ifu*, *rs* and *rob* modeling the three components of the dynamic machine have the following types (defined formally in 3.2.2):

$$\begin{aligned} \mathit{ifu} &\colon \texttt{TransSig}, \texttt{BoolSig} \to \texttt{TransSeqSig} \\ \mathit{rob} &\colon \texttt{TransSeqSig}, \texttt{TransSetSig} \to \texttt{TransSeqSig}, \texttt{TransSetSig}, \texttt{TransSig}, \texttt{BoolSig} \\ \mathit{rs} &\colon \texttt{TransSetSig}, \texttt{BoolSig} \to \texttt{TransSetSig} \end{aligned}$$

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	$\cdots$
σ_0									
σ_1	$\sigma_1[1]$	$\sigma_1[2]$	$\sigma_1[3]$	$\sigma_1[4]$					
σ_2	$\sigma_2[1]$	$\sigma_2[2]$	$\sigma_2[3]$	$\sigma_2[4]$					
σ_3	$\sigma_3[1]$	$\sigma_3[2]$	$\sigma_3[3]$	$\sigma_3[4]$					
σ_4	$\sigma_4[1]$	$\sigma_4[2]$	$\sigma_4[3]$	$\sigma_4[4]$	$\sigma_4[5]$	$\sigma_4[6]$	$\sigma_4[7]$		
σ_5	$\sigma_5[1]$	$\sigma_5[2]$	$\sigma_5[3]$	$\sigma_5[4]$	$\sigma_5[5]$	$\sigma_5[6]$	$\sigma_5[7]$	$\sigma_5[8]$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 2: Example history of computational state

The following sections contain axiomatic specifications for *ifu*, *rs* and *rob*. Later in the paper we prove that any dynamic machine satisfying these specifications reproduces the result sequence `standard` of the standard machine. This is our main theorem.

Theorem. $\langle \mathbf{pc}, \mathbf{res}\rangle(\mathtt{retired}_1\,\mathtt{retired}_2 \cdots) = \mathtt{standard}$.

Here $\mathtt{retired}_1\,\mathtt{retired}_2 \cdots$ is the concatenation of finite sequences $\mathtt{retired}_1$, $\mathtt{retired}_2$, etc.

3.2.1 Computational state

Our approach to proving top-level specifications of complex machines is based on the idea of using transactions to explicitly describe the current state of computation at any cycle. We found it convenient to represent the state of computation at the cycle n as a sequence $\sigma_n = t_1, t_2, \ldots$, consisting of transactions that have been considered since the beginning of computation. Thus, we start with $\sigma_0 = \emptyset$, and if $\sigma_n = t_1, t_2, \ldots, t_k$, then $\sigma_{n+1} = t'_1, t'_2, \ldots, t'_k, t_{k+1}, \cdots, t_l$, where the transactions $t'_1, t'_2, \ldots$ are descendants of $t_1, t_2, \ldots$ and $t_{k+1}, \ldots, t_l$ is the package of freshly fetched transactions. Of course, $t'_i = t_i$ is possible for some i; $t'_i \neq t_i$ means that the computation of the i^{th} instruction has made progress in the last cycle. It is clear that understanding the passage from σ_n to σ_{n+1} means understanding the way the machine works. In our example, the computational state is conveniently used as the ROB state, but different variations are possible.

It is appropriate to picture all sequences $\sigma_0, \sigma_1, \sigma_2, \ldots$ as rows of a table, as in the example in Figure 2. Then each column of the table is an infinite sequence $\sigma_m[i], \sigma_{m+1}[i], \sigma_{m+2}[i], \ldots$ where m corresponds to the cycle when the i^{th} instruction (I_i in Figure 2) was fetched. This is the personal history of an instruction as it goes through the stepwise computation. A useful observation is that there are only finitely many essentially different possibilities where $\sigma_n[i] \neq \sigma_{n+1}[i]$. That is, there are only finitely many "elementary" steps involved in the computation of a single instruction. We make this explicit by using the status letters to describe what stage of computation a transaction is in. If status letters are correctly chosen, then every change from $\sigma_n[i]$ to $\sigma_{n+1}[i]$ is recorded in a change of status letter. Thus, the "status words" $\tau_n = \mathbf{sts}(\sigma_n)$ and transitions from τ_n to τ_{n+1} suffice to present much of the qualitative analysis of the computation.

For example, the status words are always in the form $\{R, D\}^*\{A, C, E\}^*N^*$, which means that the computational state consists of a sequence of retired (R) and dropped (D) instructions followed by a sequence of currently active (A) and computed but not retired (C, E) instructions, followed by a sequence of instructions whose computation in the RS has not yet begun (N). The ROB axiomatics gives a precise description of what changes can occur in the passage from σ_n to σ_{n+1}.

Since the computation of every instruction takes finite time, the axioms should make sure that every sequence $\sigma_n[i]$ for fixed i stabilizes (becomes constant eventually). This in turn defines the "limit state" σ_∞, where $\sigma_\infty[i]$ is defined as the limit of $\sigma_n[i]$ as $n \to \infty$. The limit state is a convenient way of representing the result of the whole infinite computation. For example, the axioms of the components of the machine should be powerful enough to ensure that every transaction in the limit state is either retired or dropped as part of a mispredicted branch, and also that the subsequence of retired transactions essentially coincides with the result sequence of the standard machine.

3.2.2 Miscellaneous sorts and notations

It is convenient to define the sorts `TransSeq` (sequence of transactions) and `TransSet` (set of transactions) with a requirement that none of their members can contain two transactions with the same name. We also write the sort `TransSeqSig` for the sort of signals of `TransSeq`, and analagously define the sorts `TransSetSig` and `BoolSig`. The functions **pc**, ..., **sts** extend naturally to `TransSeq` and `TransSet`; for example, $\mathbf{pc}(t_1 \cdots t_n) = \mathbf{pc}(t_1) \cdots \mathbf{pc}(t_n)$ and $\mathbf{pc}\{t_1, \ldots, t_n\} = \{\mathbf{pc}(t_1), \ldots, \mathbf{pc}(t_n)\}$.

The constituent sorts and operations of the standard machine will occur in the specification of the dynamic machine as well. We shall also need a default transaction $t_{()}$ whose only property required is $\mathbf{res}(t_{()}) = \mathtt{defaultValue}$.

Suppose $\sigma = t_1 \cdots t_k \in$ `TransSeq`. If $(p, q) = \mathtt{getSources}(\mathbf{pc}(t_i), \mathbf{pc}(t_1 \cdots t_{i-1}))$ we say that t_p and t_q are the first and the second *transaction sources* of t_i in σ. If, in addition, $\mathbf{ops}(t_i) = (\mathbf{res}(t_p), \mathbf{res}(t_q))$, then we say that t_i *has correct operands in* σ.

We define $\mathtt{nextpc}(t) = \mathtt{next}(\langle \mathbf{pc}, \mathbf{res}\rangle(t))$ and say that a transaction t is *faulty* if $\mathbf{spc}(t) \neq \mathtt{nextpc}(t)$.

We shall use the notation $\alpha[i]$ for the i^{th} element of the transaction sequence α and $\alpha\langle n\rangle$ for its element whose name is n. Similarly, for a transaction set S, its member whose name is n will be denoted $S\langle n\rangle$. The empty sequence and the empty set will be both denoted $\emptyset$, and the length of a sequence α will be denoted by $|\alpha|$.

Two useful functions `replace` and `diff` are defined for both transaction sets and transaction sequences. If each of X and Y is either a transaction set or a transaction sequence, then $\mathtt{replace}(X, Y)$ is obtained from X by replacing elements $X\langle i\rangle$ with $Y\langle i\rangle$ for every i for which it is possible, and $\mathtt{diff}(X, Y)$ is obtained by removing from X all elements $X\langle i\rangle$ such that $X\langle i\rangle = Y\langle i\rangle$.

For $\sigma \in$ `TransSeq` and $S \subset$ `Status`, define σ^S to be the subsequence of σ consisting of all transactions whose status belongs to S. We will shorten this notation and, for example, write σ^{RD} instead of $\sigma^{\{R,D\}}$. Define also $\sigma^\circ = \sigma^S$, where $S = \{R, A, C, E\}$.

3.3 IFU specification

In the IFU axioms below we assume that `lastRet`, `error` and `fetched` are signals satisfying the relation $\mathit{ifu}(\texttt{lastRet}, \texttt{error}) = \texttt{fetched}$.

Axiom IFU-1. If $\texttt{fetched}_n = t_1 \cdots t_p$ and $\texttt{fetched}_i = \emptyset$ for all $i < n$, then $\mathbf{pc}(t_1) = \texttt{startIdx}$.

Axiom IFU-2. If $\texttt{fetched}_n = t_1 \cdots t_p$, then $\mathbf{pc}(t_i) = \mathbf{spc}(t_{i-1})$ for every $i \in \{2, \ldots, p\}$.

Axiom IFU-3. Let $m < n$, $\texttt{fetched}_m = t_1 \cdots t_p$, $\texttt{fetched}_n = t'_1 \cdots t'_q$, and $\texttt{fetched}_i = \emptyset$ for i between m and n. Then:
(a) If $\texttt{error}_i$ is false for all i such that $m \le i < n$, then $\mathbf{pc}(t'_1) = \mathbf{spc}(t_p)$;
(b) If i is the smallest integer such that $m \le i < n$ and $\texttt{error}_i$ is true, then $\mathbf{pc}(t'_1) = \texttt{nextpc}(\texttt{lastRet}_i)$.

Axiom IFU-4. For every m there exists n such that $n > m$ and $\texttt{fetched}_n \neq \emptyset$.

The axioms are conditions that the function *ifu* is required to satisfy. Axiom IFU-1 states that when the IFU fetches the first instruction, it will fetch from `startIdx`. Axiom IFU-2 indicates that the IFU fetches "consecutive" instructions, and defines "consecutive" for branch instructions to be the instruction pointed to by the branch's speculative program counter. Axiom IFU-3 clarifies the relationship between two consecutive non-empty fetches. If the `error` signal was set at the time of the first fetch or in the meantime, then the first instruction of the second fetch should be the correct successor of the last retired instruction. Otherwise (when there are no errors between the two fetches), speculative fetching continues. Finally, Axiom IFU-4 simply states that fetching never ceases.

3.4 RS specification

In the RS axioms we assume the signals `ready`, `error`, and `computed` satisfy the equality $\mathit{rs}(\texttt{ready}, \texttt{error}) = \texttt{computed}$. We use two sets $\texttt{contents}_n$ and $\texttt{justComputed}_n$ to describe the state of the RS. The set $\texttt{contents}_n$ is meant to contain the transactions present in the RS at the n^{th} cycle, and $\texttt{justComputed}_n$ corresponds to a subset of $\texttt{computed}_n$ whose elements have "correct" **res** components. We also need an auxiliary function $\texttt{updtOps}\colon \texttt{TransSet} \to \texttt{TransSet}$ whose effect is to replace all "reference" operands of the form $\mathbf{name}(s)$ with the available values $\mathbf{res}(s)$.

Definition 1. $\texttt{updtOps}(S) = \{f(t, S) \mid t \in S\}$, *where* $f\colon \texttt{Trans}, \texttt{TransSet} \to \texttt{Trans}$ *is the function defined by:* $f(t, S) = t'$ *if and only if* t *and* t' *have the same components except* **ops**, *and*

$$\mathbf{op1}(t') = \begin{cases} \mathbf{res}(s) & \textit{if } \mathbf{op1}(t) = \mathbf{name}(s) \textit{ for some } s \in S \\ \mathbf{op1}(t) & \textit{otherwise} \end{cases}$$

and similarly for **op2**.

Axiom RS-1. $\texttt{computed}_n \subset \{t \mid t \in \texttt{contents}_n \text{ and } \mathbf{res}(t) \in \texttt{Value}\}$.

Axiom RS-2. For every $t \in \texttt{justComputed}_n$ there exists $s \in \texttt{contents}_n$ such that $\mathbf{res}(t) = \texttt{compute}(\langle \mathbf{pc}, \mathbf{ops} \rangle (s))$ and the other components of t are the same as those of s.

Axiom RS-3. If $n > 0$, $\texttt{error}_{n-1}$ is false, and the sets of names of transactions in $\texttt{ready}_{n-1}$ and $\texttt{contents}_{n-1}$ are disjoint, then

$$\texttt{contents}_n = \texttt{updtOps}(\texttt{ready}_{n-1} \cup \texttt{c}_{n-1}) \setminus \texttt{computed}_{n-1}$$

where $\texttt{c}_{n-1} = \texttt{replace}(\texttt{contents}_{n-1}, \texttt{justComputed}_{n-1})$. In all other cases, $\texttt{contents}_n = \emptyset$.

Axiom RS-4. If a transaction t belongs to $\texttt{contents}_n$ and if $\mathbf{op1}(t), \mathbf{op2}(t) \in \texttt{Value}$, then there exists $m > n$ such that $\texttt{contents}_m$ does not contain a transaction whose name is $\mathbf{name}(t)$.

Axiom RS-1 states that transactions returned by the RS through the computed signal have no further need of computation. It also constrains the computed signal to contain instructions that have come from somewhere, i.e. the RS cannot create hoax transactions to pass through computed. Axiom RS-2 clarifies the relationship between $\texttt{justComputed}_n$ and $\texttt{contents}_n$. Again, this axiom precludes the RS from creating new transactions with no correspondence to the state of the RS. It also states that the result of a computation in the RS should be equivalent to the result computed in the standard machine. Axiom RS-3 inductively defines the persistent state of the RS. If no exception is raised, then the contents at cycle n equals the contents at $n-1$, combined with the new instructions sent from the ROB, and minus the instructions computed at $n-1$. Also, the results of newly computed transactions are forwarded in the process to those transactions which need them as operands. Finally, Axiom RS-4 states that each instruction that is present in the RS and has values as operands will eventually disappear from RS—it will either be passed back through the computed wire, or squashed if error occurs in the future.

3.5 ROB specification

We treat the ROB as a state machine by specifying the function

$$rob' \colon \texttt{TransSeq}, \texttt{TransSet}, \texttt{State} \to \texttt{State}, \texttt{TransSeq}, \texttt{TransSet}, \texttt{Trans}, \texttt{Boolean}.$$

Precisely, the equality $rob(\texttt{fetched}, \texttt{computed}) = (\texttt{retired}, \texttt{ready}, \texttt{lastRet}, \texttt{error})$ holds if and only if, for every $n \geq 0$, the equality $rob'(\texttt{fetched}_n, \texttt{computed}_n, \texttt{state}_n) = (\texttt{state}_{n+1}, \texttt{retired}_n, \texttt{ready}_n, \texttt{lastRet}_n, \texttt{error}_n)$ holds. The axioms below are stated as conditions on $\texttt{fetched}_n$, $\texttt{computed}_n$, $\texttt{state}_n$, $\texttt{state}_{n+1}$, $\texttt{retired}_n$, $\texttt{ready}_n$, $\texttt{lastRet}_n$, $\texttt{error}_n$.

As indicated in Subsection 3.2.1, the state of the ROB contains all transactions ever considered. We put them in a sequence respecting the order of fetching. The *status word* $\mathsf{sts}(\sigma)$ of any state σ always has a certain form and it is convenient to put that restriction into the sort definition:

$$\texttt{State} = \{\sigma \in \texttt{TransSeq} \mid \mathsf{sts}(\sigma) \in \{R,D\}^*\{A,C,E\}^*N^*\}.$$

The specification of the ROB uses four auxiliary functions:

$$\begin{aligned} \texttt{acceptFtchd}&\colon \texttt{State}, \texttt{TransSeq} \to \texttt{State} \\ \texttt{getOps}&\colon \texttt{State} \to \texttt{State} \\ \texttt{acceptCptd}&\colon \texttt{State}, \texttt{TransSet} \to \texttt{State} \\ \texttt{retire}&\colon \texttt{State} \to \texttt{State} \end{aligned}$$

Definition 2. $\texttt{acceptFtchd}(\sigma,\alpha) = \sigma\beta$, *where* $|\beta| = |\alpha|$ *and for every* i *such that* $1 \leq i \leq |\beta|$
(a) $\mathbf{name}(\beta[i]) = \mathbf{res}(\beta[i]) = |\sigma| + i$;
(b) $\mathsf{sts}(\beta[i]) = N$;
(c) The remaining components of β *equal the corresponding components of* α.

Definition 3. $\texttt{getOps}(\sigma) = \omega$ *if and only if* ω *is obtained from* σ *by replacing the first transaction* t *of* σ *whose status is* N *(if it exists) with a transaction* t' *so that* t' *has status* A *and correct operands in* ω, *and has all other components same as* t.

Definition 4. *Given* $S \in \texttt{TransSet}$ *and* $\sigma \in \texttt{State}$, *let* S' *consist of all transactions* t *of* S *whose status letter is modified so that* $\mathsf{sts}(t)$ *is* E *or* C *depending on whether* t *is faulty or not. Define* $\texttt{acceptCptd}(\sigma,S) = \texttt{replace}(\sigma,S')$.

Definition 5. *Define* $\texttt{retire}'\colon \texttt{Status}^* \to \texttt{Status}^*$ *as follows. For* $\tau \in \texttt{Status}^*$, *let* τ^{RD} *be the maximal prefix of* τ *which uses only letters* R, D. *If* $\tau = \tau^{RD}C\theta$, *then* $\texttt{retire}'(\tau) = \tau^{RD}R\theta$. *If* $\tau = \tau^{RD}E\theta$, *then* $\texttt{retire}'(\tau) = \tau^{RD}RD^{|\theta|}$. *In all other cases,* $\texttt{retire}'(\tau) = \tau$.

Definition 6. $\texttt{retire}(\sigma) = \omega$ *if and only if* $\mathsf{sts}(\omega) = \texttt{retire}'(\mathsf{sts}(\sigma))$ *and all other components of* ω *are equal to the corresponding components of* σ.

Axiom ROB-1. The initial state $\texttt{state}_0$ is empty. For every $n > 0$, there exist $\xi_1, \xi_2, \xi_3 \in \texttt{State}$ and integers $k \geq 0$ and $l \geq 1$ such that

$$\begin{aligned} \xi_1 &= \texttt{acceptFtchd}(\texttt{state}_n, \texttt{fetched}_n) \\ \xi_2 &= \texttt{getOps}^k(\xi_1^\circ) \\ \xi_3 &= \texttt{acceptCptd}(\xi_2, \texttt{computed}_n) \\ \texttt{state}_{n+1} &= \texttt{retire}^l(\xi_3) \end{aligned}$$

Axiom ROB-2. With ξ_1, ξ_2, ξ_3 as in Axiom ROB-1,
(a) $\texttt{ready}_n$ is the set determined by the sequence $\texttt{diff}(\xi_2, \xi_1)$;
(b) $\texttt{retired}_n = \texttt{diff}(\texttt{state}^R_{n+1}, \xi^R_3)$;
(c) If $\texttt{retired}_n \neq \emptyset$, then $\texttt{lastRet}_n$ is the last element in $\texttt{retired}_n$;
(d) $\texttt{error}_n = \mathit{true}$ if and only if $\texttt{retired}_n \neq \emptyset$ and $\texttt{lastRet}_n$ is faulty.

Axiom ROB-3. If $\texttt{state}^A_n = \emptyset$ and $\texttt{state}^N_n \neq \emptyset$, then $\texttt{ready}_n \neq \emptyset$.

The axiom ROB-1 deserves a detailed explanation. To shorten the notation, let us use σ_n for $\texttt{state}_n$, $\tau_n = \texttt{sts}(\sigma_n)$, and $\rho_n = \texttt{retired}_n$. We want to look closely at the transitions from σ_n to σ_{n+1} and from τ_n to τ_{n+1} through three intermediate stages. Let ζ_1, ζ_2, ζ_3 be the status words of the intermediate states ξ_1, ξ_2 and ξ_3 of ROB-1. Suppose $\tau_n = \tau_n^{RD}\omega N^p$. Then $\zeta_1 = \tau_n^{RD}\omega N^{p+q}$, where $q = |\texttt{fetched}_n|$. Note that σ_n survives intact as a prefix of ξ_1. Now $\zeta_2 = \tau_n^{RD}\omega A^k N^{p+q-k}$, for some k (equal to the length of $\texttt{ready}_n$). Clearly, $\xi_2[i] \neq \xi_1[i]$ implies $\zeta_1[i] = N$ and $\zeta_2[i] = A$. As a result of the next step, $\zeta_3 = \tau_n^{RD}\omega' N^{p+q-k}$, where ω' is the result of replacing some A's in ωA^k with C or E. Again, every change is recorded in a change of status letter: if $\xi_3[i] \neq \xi_2[i]$, then $\zeta_2[i] = A$ and $\zeta_3[i]$ is C or E.

The last passage, from ζ_3 to τ_{n+1}, is the most complicated. If the first letter of ω' is not C or E, then $\zeta_3 = \tau_{n+1}$ and $\xi_3 = \sigma_{n+1}$. Otherwise, we can write $\omega' = C^{r-1}E\omega''$ or $\omega' = C^r\omega''$, where $r \geq 1$ and, in the second case, ω'' does not begin with C or E. Let us call this prefix $C^{r-1}E$ or C^r of ω' the *critical segment* of ζ_3. What happens now is that letters of some prefix of the critical segment get replaced with R. If $l < r$, that is all that happens, but if $l = r$ and if the critical segment is $C^{r-1}E$, then, aside from the transformation of the critical segment into R^r, a dramatic change occurs to the right of the critical segment—all its letters get replaced with D. This special situation requires a special treatment in many arguments that follow, so we shall refer to n as being *singular* if the change from σ_n to σ_{n+1} involves this "flushing" of transactions. Otherwise, n will be called *regular.* Note that $\texttt{retired}_n$ is the subsequence of ξ_3 that corresponds to the subword of ζ_3 that is replaced with R^r. We still need to consider the case when $l > r$, but it brings nothing new, since, as we can easily check, $\texttt{retire}^l(\xi_3) = \texttt{retire}^r(\xi_3)$ if $l > r$. Observe finally that $\sigma_{n+1}[i] \neq \xi_3[i]$ implies that either $\tau_{n+1}[i] = R$ and $\zeta_3[i]$ is C or E, or $\tau_{n+1}[i] = D$ and $\zeta_3[i]$ is A, C, E or N.

4 Correctness proof

So far we have axiomatized the individual units. Now we demonstrate that the axiomatization is sufficient to obtain a global correctness property.

In addition to shorthands σ_n, τ_n, we will also use ρ_n for $\texttt{retired}_n$. Recall that our goal is to show $\langle \mathbf{pc}, \mathbf{res}\rangle(\rho_\infty) = \texttt{standard}$, where $\rho_\infty = \rho_1\rho_2\cdots$.

4.1 State transitions and the limit state

Lemma 1 (Name is index). *Whenever defined, $\sigma_n[i] = \sigma_n\langle i\rangle$. Consequently,* $\mathbf{name}(\sigma_n)$ *is an initial segment of the sequence of positive integers.*

Proof. This is a matter of checking the property $\mathbf{name}(\sigma[i]) = i$ for all states $\sigma = \sigma_n$. The empty state obviously has that property. Arguing by induction, assume σ_n has the property. We use ROB-1 and its notation. The first thing to observe is that ξ_1 has our property by Definition 2. Then we have $\mathbf{name}(\xi_1) = \mathbf{name}(\xi_2) = \mathbf{name}(\xi_3) = \mathbf{name}(\sigma_{n+1})$, where the three equalities are justified by Definitions 3,4, 6 respectively. Thus, σ_{n+1} has the property considered. □

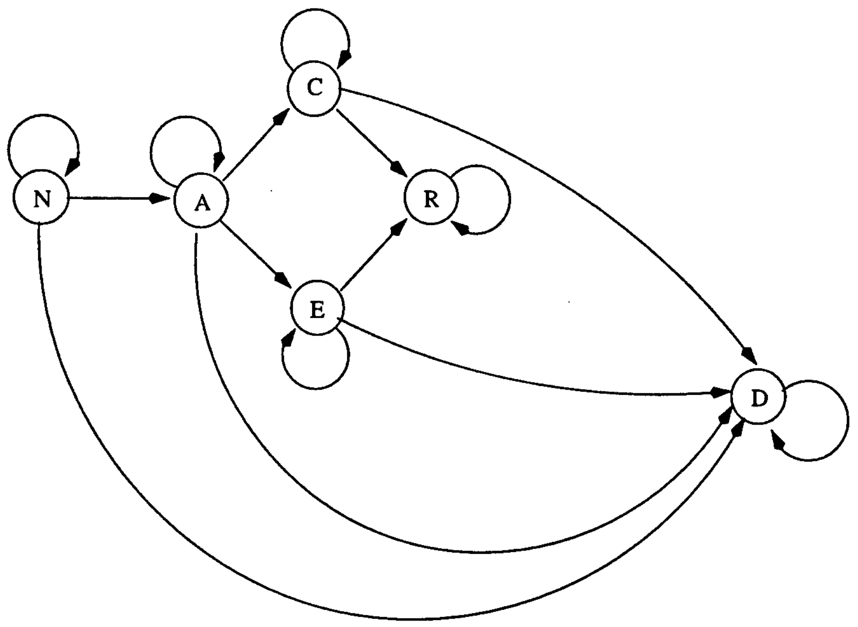


Figure 3: Transaction status transition diagram

Lemma 2 (Retired). *If n is regular, then $\sigma_{n+1}^{RD} = \sigma_n^{RD}\rho_n$. If n is singular, then $\rho_n \neq \emptyset$ and $\sigma_{n+1} = \sigma_{n+1}^{RD} = \sigma_n^{RD}\rho_n\delta_n$, for some δ_n such that* $\mathbf{sts}(\delta_n) \in D^*$. □

It follows that $\sigma_{n+1}^{R} = \sigma_n^{R}\rho_n$ and so $\rho_\infty = \lim \sigma_n^R$. However, at this point it is not even clear that ρ_∞ is an infinite sequence.

Every change in the four step transition from $\sigma_n[i]$ to $\sigma_{n+1}[i]$ is reflected in a change of status letter. The following lemma states this precisely.

Lemma 3 (State transition). *If $\sigma_{n+1}[i] \neq \sigma_n[i]$, then $\tau_{n+1}[i] \neq \tau_n[i]$. The arrows in the transition diagram in Figure 3 correspond to all possible pairs $(\tau_n[i], \tau_{n+1}[i])$.* □

Lemma 4. *If $\tau_n[i] \in \{R, C, E\}$, then* $\mathbf{res}(\sigma_n[i]) \in \texttt{Value}$. □

Lemma 5 (Faulty). *If $\tau_n[i] = E$ then t is faulty, and if $\tau_n[i] = C$ then is not faulty.*

Proof. Suppose $\tau_n[i]$ is E or C. Let $m+1$ be the smallest integer such that $\tau_{m+1}[i] = \tau_n[i]$ and let $t = \sigma_{m+1}[i]$. By Lemma 3, $\sigma_n[i]$ and t are at the same time faulty or not. If ξ_1, ξ_2, ξ_3 are the intermediate states in the transition from σ_m to σ_{m+1}, it follows from the the commentary to ROB-1 that $\zeta_3[i] = \mathbf{sts}(t) = \tau_n[i]$ and $\zeta_2[i] = A$. By Lemma 1, $\xi_3[i] = \xi_3\langle i\rangle$, and, by Definition 4, $\xi_3\langle i\rangle$ is obtained from $t' = \texttt{computed}_m\langle i\rangle$, and t is faulty or not depending on whether $\zeta_3[i]$ is C or E. Now t and t' are at the same time faulty or not because the passage from ζ_3 to σ_{m+1} does not affect the components **pc**, **spc**, **res** in terms of which being faulty is defined. □

Lemma 6 (Faulty retired). *Suppose t is a transaction occurring in some ρ_n. Then t is faulty if and only if n is singular and t is the last transaction in ρ_n.*

Proof. Referring again to the commentary to ROB-1, $\mathbf{sts}(\rho_n) = R^r$ corresponds to the subword $C^{r-1}E$ or C^r of ζ_3, depending on whether n is singular or regular. The desired result follows then from Lemma 5. □

Lemma 7 (Error). *An integer n is singular if and only if* $\texttt{error}_n$ *is true.*

Proof. If $\rho_n = \emptyset$, then n is regular (Lemma 2) and $\texttt{error}_n$ is false (ROB-2). Assume then ρ_n is non-empty. By ROB-2, we have $\texttt{error}_n$ is true if and only if the last element of ρ_n is faulty. Lemma 6 then finishes the proof. □

We turn now to the whole personal history of an instruction as it goes through the execution process, that is, the sequence $\sigma_n\langle i\rangle$ for fixed i. Recall that $\sigma_n[i] = \sigma_n\langle i\rangle$ so that the instruction named i remains in the i^{th} place in all states σ_n in which it occurs.

The ROB axioms imply that $|\sigma_{n+1}| = |\sigma_n| + |\texttt{fetched}_n|$, and it follows then from IFU-4 that $\lim |\sigma_n| = \infty$. Moreover, for any i there exist n such that $|\sigma_{n-1}| < i \leq |\sigma_n|$, and thus $\sigma_m[i]$ exists if and only if $m \geq n$.

Lemma 8 (Stabilization). *For fixed i, the sequence $\sigma_n[i]$ is eventually constant.*

Proof. In view of Lemma 3, it suffices to show that the sequence $\tau_n[i]$ is eventually constant. Indeed, the sequence $\tau_n[i]$ corresponds to a (directed) path in the graph of Figure 3, and that graph has no non-trivial loops. □

Since each sequence $\sigma_n[i]$ (for fixed i) stabilizes, there exists a *limit state* $\sigma_\infty = \lim \sigma_n$. It follows immediately that the sequence τ_n of status words has a limit τ_∞, and that $\tau_\infty = \mathbf{sts}(\sigma_\infty)$. Since $\lim |\sigma_n| = \infty$, the limit state σ_∞ is an infinite sequence.

Since σ_n^{RD} is a prefix of σ_n, it follows that $\lim \sigma_n^{RD}$ is a prefix of σ_∞. We would like to show that the two are in fact equal or, equivalently, that $\sigma_\infty = \sigma_\infty^{RD}$. This means that every transaction is eventually retired or dropped, and we need to derive some results about the interaction between ROB and RS in order to prove that.

4.2 In the Reservation Station

From now on we shall use a shorthand S_n for $\texttt{contents}_n$. The first result shows that ROB has information about the contents of RS.

Lemma 9 (RS-contents). $\mathbf{name}(S_n) = \mathbf{name}(\sigma_n^A)$.

Proof. We argue by induction, the case $n = 0$ being trivial since, by definition, both S_0 and σ_0 are empty. Assume then the lemma is true for some n. If n is singular, then $\sigma_{n+1}^A = \emptyset$ because $\sigma_{n+1} = \sigma_{n+1}^{RD}$ (Lemma 2), and $\texttt{contents}_{n+1} = \emptyset$ by RS-3 and Lemma 7. It remains to consider the case when n is regular. By RS-1,

$\mathtt{computed}_n \subset \mathtt{contents}_n$. By induction, $\mathbf{name}(S_n) \subset \cup_{i<n}\mathbf{name}(\mathtt{ready}_i)$. Since the sets $\mathbf{name}(\mathtt{ready}_i)$ are pairwise disjoint, we obtain from RS-3 that

$$\mathbf{name}(S_{n+1}) = (\mathbf{name}(S_n) \cup \mathbf{name}(\mathtt{ready}_n)) \setminus \mathbf{name}(\mathtt{computed}_n).$$

We prove now that $\mathbf{name}(\sigma_{n+1}^A)$ and $\mathbf{name}(\sigma_n^A)$ satisfy the same relation. Using the notation of ROB-1, we have $\mathbf{name}(\xi_1^A) = \mathbf{name}(\sigma_n^A)$ by definition of `acceptFtchd`. Then $\mathbf{name}(\xi_2^A) = \mathbf{name}(\xi_1^A) \cup \mathbf{name}(\mathtt{ready}_n)$ by definition of `getOps`. Then $\mathbf{name}(\xi_3^A) = \mathbf{name}(\xi_2^A) \setminus \mathbf{name}(\mathtt{computed}_n)$ by definition of `acceptCptd`. Finally, $\mathbf{name}(\sigma_{n+1}^A) = \mathbf{name}(\xi_3^A)$ because the transition from ξ_3 to σ_{n+1} only involves status letter changes from C to R when n is regular. Thus,

$$\mathbf{name}(\sigma_{n+1}^A) = (\mathbf{name}(\sigma_n^A) \cup \mathbf{name}(\mathtt{ready}_n)) \setminus \mathbf{name}(\mathtt{computed}_n). \square$$

Let $\mathtt{span}(i)$ denote the set of all n such that $i \in S_n$. By the previous lemma, $\mathtt{span}(i)$ can be viewed as the set of all n such that $\tau_n[i] = A$. Suppose $\mathtt{span}(i)$ is non-empty. Then it is a set of consecutive integers; there exists n such that $i \in \mathbf{name}(\mathtt{ready}_n)$, and $n+1$ is the smallest element of $\mathtt{span}(i)$. The maximal element of $\mathtt{span}(i)$ (if it exists!) is that n for which $\mathtt{error}_n$ is true or $\mathtt{computed}_n$ exists. All these facts follow easily from the relation between $\mathbf{name}(S_n)$ and $\mathbf{name}(S_{n+1})$ established in the proof of Lemma 9.

In view of definition of `updtOps` and *RS*-axioms, the evolution of a transaction in RS affects only the **res** and **ops** components. The following lemma makes this observation more precise. To state it properly, we introduce the concept of *RS-status* of transactions. We define $\mathtt{rs\text{-}sts}(t)$ to be a three-letter word xyz, where the $x, y, z \in \{n, v\}$. The letter z is defined to be n if $\mathbf{res}(t) \in \mathtt{Name}$, and to be v if $\mathbf{res}(t) \in \mathtt{Value}$. In the same manner, $\mathbf{op1}(t)$ and $\mathbf{op2}(t)$ determine the values of x and y respectively.

The formula in the axiom RS-3 implies that there are three steps in the transition from S_n to S_{n+1} when n is regular:

$$\begin{aligned} S_n &\longmapsto S' = \mathtt{replace}(S_n, \mathtt{justComputed}_n) \\ &\longmapsto S'' = \mathtt{updtOps}(\mathtt{ready}_n \cup S') \\ &\longmapsto S_{n+1} = S'' \setminus \mathtt{computed}_n. \end{aligned}$$

If one of $S_n\langle i\rangle$, $S_{n+1}\langle i\rangle$ exists and the other does not, that is the responsibility of $\mathtt{ready}_n$ or $\mathtt{computed}_n$, as we have already observed. Suppose both exist. Then $S'\langle i\rangle$ and $S''\langle i\rangle$ exist and the later is equal to $S_{n+1}\langle i\rangle$. Thus, if $S_n\langle i\rangle \neq S_{n+1}\langle i\rangle$, then either $S_n\langle i\rangle \neq S'\langle i\rangle$ or $S'\langle i\rangle \neq S_{n+1}\langle i\rangle$ (or both, which we will see shortly does not happen). Suppose $t = S_n\langle i\rangle$ and $t' = S'\langle i\rangle$ are not equal. By definition of `replace`, we have $t' \in \mathtt{justComputed}_n$, and the axiom RS-2 implies $\mathtt{rs\text{-}sts}(t') = vvv$. Axiom RS-2 also gives us $\mathbf{ops}(t) = \mathbf{ops}(t')$, so $\mathtt{rs\text{-}sts}(t)$ is either vvn or vvv. We will prove that the latter case does not occur. The second case to consider is when $t' = S'\langle i\rangle$ and $t'' = S_{n+1}\langle i\rangle$ are non-equal. This is an application of `updtOps`, and it follows that one or both operands of t' are names and the corresponding operands of t'' are values. It follows immediately that one cannot have both $t \neq t' \neq t''$—one inequality requires t' to have value operands, the other asks for at least one name operand.

Let us look at the whole set $\mathtt{span}(i)$; denote $t_k = S_k\langle i\rangle$. Suppose $m = \min \mathtt{span}(i)$. Then there exists $t \in \mathtt{ready}_{m-1}$ such that t_m has all components equal to those

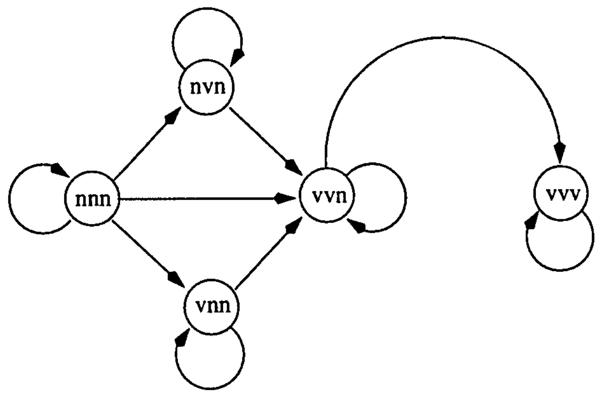


Figure 4: Transaction values transition diagram

of t except perhaps **ops**. Since $\textbf{res}(t) = \textbf{name}(t) = i$ (by definition of `ready`) it follows that $\textbf{res}(t_m) = i$. Consider now two consecutive members t_n and t_{n+1} of the sequence $t_m, t_{m+1}, \ldots$ As shown in the previous paragraph, if $\textbf{res}(t_n) \neq \textbf{res}(t_{n+1})$, then $\textbf{res}(t_n)$ is a name (necessarily i), and $\textbf{res}(t_{n+1})$ is a value. Referring again to the paragraph above, it follows that $\texttt{rs-sts}(t_n) = vvn$ and $\texttt{rs-sts}(t_{n+1}) = vvv$. It also follows that $t_k = t_{n+1}$ for all $k > n$, so $\textbf{ops}(t_k) \neq \textbf{ops}(t_{k+1})$ is possible only when $\textbf{res}(t_k) = \textbf{res}(t_{k+1}) \in \texttt{Name}$.

Summarizing, we have the following.

Lemma 10 (RS-transition). *Let* $t = S_n\langle i \rangle$ *and* $t' = S_{n+1}\langle i \rangle$. *Then* $\langle \textbf{pc}, \textbf{spc} \rangle(t) = \langle \textbf{pc}, \textbf{spc} \rangle(t')$. *The inequality* $t \neq t'$ *occurs only when* $\texttt{rs-sts}(t) \neq \texttt{rs-sts}(t')$. *The transition diagram in Figure 4 describes all possible pairs* $(\texttt{rs-sts}(t), \texttt{rs-sts}(t'))$. □

Corollary 1. *If* t, t' *are transactions in* σ_n *and* S_n *respectively, and if* $\textbf{name}(t) = \textbf{name}(t')$, *then* $\langle \textbf{pc}, \textbf{spc} \rangle(t) = \langle \textbf{pc}, \textbf{spc} \rangle(t')$. □

Now we need to deal with the simultaneous evolution of a transaction and its sources. Suppose $\textsf{span}(i)$ is non-empty, let n be its minimum, and $t_0 \in \texttt{ready}_{n-1}$ be the element giving rise to $S_n\langle i \rangle$, as in the paragraph before Lemma 10. By definition of `ready`, $\textbf{op1}(t_0)$ is either `defaultValue` or $\textbf{res}(\xi_1[p])$ for some $p < i$ determined by the function **getSources**, where ξ_1 is the first intermediate state between σ_{n-1} and σ_n; see ROB-1. Let us say that p is the *first source* of i, and similarly define the *second source* of i (if it exists).

Lemma 11 (Forwarding). *Suppose* $t = S_n\langle i \rangle$ *and* p *is the first source of* i.
(a) If $S_n\langle p \rangle$ *does not exist, then* $\textbf{op1}(t) = \textbf{res}(\sigma_n[p]) \in \texttt{Value}$.
(b) If $s = S_n\langle p \rangle$ *exists, then* $\textbf{op1}(t) = \textbf{res}(s)$ *(which is either in* `Name` *or in* `Value`*).*
(Analogous results hold for the second source/operands.)

Proof. The proof is by induction on $n \in \textsf{span}(i)$. The difficult part is the initial case $n = \min \textsf{span}(i)$. Let $t_0 \in \texttt{ready}_{n-1}$ be as above. Recall that all components of t_0 and t are the same except possibly that some operand of t_0 is in `Name`, and

the corresponding operand of t is in $\texttt{Value}$. We consider three cases separately. Let $S' = \texttt{replace}(S_{n-1}, \texttt{justComputed}_{n-1})$ and $S'' = \texttt{updtOps}(\texttt{ready}_{n-1} \cup S')$. We have $t = S''\langle i\rangle$ and $\mathbf{op1}(t) = \mathbf{res}(S'\langle p\rangle)$ if $S'\langle p\rangle$ exists; otherwise $\mathbf{op1}(t) = \mathbf{op1}(t_0)$. The proof splits into three cases.
Case 1: $\mathbf{op1}(t) \in \texttt{Name}$. We prove that $S_n\langle p\rangle$ necessarily exists and that $\mathbf{op1}(t) = p = \mathbf{res}(S_n\langle p\rangle)$; that will prove the lemma in the case considered.

First we have $\mathbf{op1}(t) = \mathbf{op1}(t_0) = \mathbf{res}(\xi_1[p]) = p$. We claim that $\sigma_n[p] = \xi_1[p]$. Indeed, $\xi_2[p] = \xi_1[p]$ is obvious, and $\sigma_{n+1}[p] \neq \xi_2[p]$ is possible only if $p \in \mathbf{name}(\texttt{computed}_n)$. This would imply that $\mathbf{res}(S_n\langle p\rangle) \in \texttt{Value}$, and then that $\mathbf{res}(t) \in \texttt{Value}$, which is not true. Thus, $\mathbf{res}(\sigma_n[p])$ is not in $\texttt{Value}$, and by Lemma 4, $\tau_n[p] \notin \{R, C, E\}$. By definition of $\texttt{ready}$, $\mathbf{sts}(\xi_1[p]) \neq D$ and this implies $\tau_n[p] \neq D$, because no new elements with status D arise in transition from σ_{n-1} to σ_n ($n-1$ is regular as $S_n \neq \emptyset$). Since $\tau_n[i] = A$ (Lemma 9) and $p < i$, we have $\tau_n[p] \neq N$. The only remaining possibility is $\tau_n[p] = A$, and so $s = S_n\langle p\rangle$ exists. It remains to prove that $\mathbf{res}(s) = p$. Assume the contrary; then $\mathbf{res}(s) \in \texttt{Value}$ and $S_{n-1}\langle p\rangle$ exists. Moreover, we have either $s' = s$ or $s' \in \texttt{justComputed}_{n-1}$. Now $S'\langle i\rangle = s'$ and by definition of $\texttt{updtOps}$, the first operand of $t = S''\langle i\rangle$ is $\mathbf{res}(s')$, contradicting $\mathbf{op1}(t) = p$.
Case 2: $\mathbf{op1}(t_0) \in \texttt{Value}$. Now $\mathbf{res}(\xi_1[p])$ is in $\texttt{Value}$ and so is equal to $\mathbf{res}(\sigma_{n-1}[p]) = \mathbf{res}(\sigma_n[p])$. By Lemma 4 $\tau_{n-1}[p] \neq A$, so $S_{n-1}\langle p\rangle$ does not exist. Thus, $\mathbf{op1}(t) = \mathbf{op1}(t_0) = \mathbf{res}(\xi_1[p]) = \mathbf{res}(\sigma_n[p])$.
Case 3: $\mathbf{op1}(t) \in \texttt{Value}$ and $\mathbf{op1}(t_0) \in \texttt{Name}$. We have $\mathbf{op1}(t) = \mathbf{res}(S'\langle p\rangle)$. If $p \notin \mathbf{name}(\texttt{computed}_{n-1})$, then $S_n\langle p\rangle$ exists and $\mathbf{op1}(t) = \mathbf{res}(S'\langle p\rangle) = \mathbf{res}(S_n\langle p\rangle)$. If $p \in \mathbf{name}(\texttt{computed}_{n-1})$, then $\mathbf{op1}(t) = \mathbf{res}(\texttt{computed}_{n-1}\langle p\rangle)$. Moreover, $S_n\langle p\rangle$ does not exist, but $\mathbf{res}(\sigma_n[p]) = \mathbf{res}(\texttt{computed}_{n-1}\langle p\rangle)$ by Definition 4, so the lemma is true in both cases.

Suppose now $n \neq \min \mathsf{span}(i)$ and the lemma is true for $n-1$. Let $t' = S_{n-1}\langle i\rangle$. If $S_{n-1}\langle p\rangle$ does not exist, then $\mathbf{op1}(t') = \mathbf{res}(\sigma_{n-1}[p]) \in \texttt{Value}$, by the induction hypothesis. But then $S_n\langle p\rangle$ does not exist either, and $\mathbf{res}(\sigma_n[p]) = \mathbf{res}(\sigma_{n-1}[p])$, proving the lemma.

Suppose now $s' = S_{n-1}\langle p\rangle$ exists. By induction hypothesis, $\mathbf{op1}(t') = \mathbf{res}(s')$. If $S_n\langle p\rangle$ does not exist, then $p \in \mathbf{name}(\texttt{computed}_{n-1})$, and we obtain $\mathbf{op1}(t) = \mathbf{res}(\sigma_n[p]) \in \texttt{Value}$ as in the corresponding situation in Case 1 above. Finally, if $s = S_n\langle p\rangle$ exists, we either have $s = s'$ which implies $\mathbf{op1}(t) = \mathbf{op1}(t') = \mathbf{res}(s') = \mathbf{res}(s)$, or $s \in \texttt{justComputed}_{n-1}$ which also implies $\mathbf{op1}(t) = \mathbf{res}(s)$. □

Axiom RS-4 implies that $\mathsf{span}(i)$ is finite if for some n both operands of $\texttt{contents}_n\langle i\rangle$ are in $\texttt{Value}$. Using the previous lemma one can show that this is true unconditionally.

Lemma 12 (Span). *For every $i \in \mathbf{N}$, the set* $\mathsf{span}(i)$ *is finite.*

Proof. We argue by induction on i. Axiom RS-4 implies that $\mathsf{span}(i)$ is finite if for some n both operands of $S_n\langle i\rangle$ are in $\texttt{Value}$. Suppose that $\mathsf{span}(j)$ is finite for all $j < i$ and that $S_n\langle i\rangle$ has at least one operand in $\texttt{Name}$, say $\mathbf{op1}(S_n\langle i\rangle) = p$. By Lemma 11(a), $S_n\langle p\rangle$ exists. By induction hypothesis, for some $m > n$, $S_m\langle p\rangle$ does not exist. Then Lemma 11(a) again implies that $\mathbf{op1}(S_m\langle i\rangle)$ is in $\texttt{Value}$. If both operands of $S_n\langle i\rangle$ are in $\texttt{Name}$, it is clear now that we can take m large enough to ensure that both operands of $S_m\langle i\rangle$ are in $\texttt{Value}$. □

4.3 Retiring never stops

Lemma 13 (No deadlock). *One has $\tau_\infty = \lim \tau_n^{RD}$, and so τ_∞ is an infinite sequence involving only letters R and D.*

Proof. Since τ_n^{RD} is a prefix of τ_{n+1}^{RD} (Lemma 2), there exists a limit $\tau^{RD} = \lim \tau_n^{RD}$. If τ^{RD} is infinite, we are done. So suppose τ^{RD} is finite. Then there is a letter $X \neq R, D$ such that $\tau_n = \tau^{RD} X \omega_n$ for all large enough n. Denote $k = |\tau^{RD}|$.

Suppose $X = A$. By Lemma 9, the set $\texttt{contents}_n$ contains a transaction named $k+1$ for all large n, which contradicts Lemma 12.

Suppose $X = N$. Then $\omega_n \in N^*$, so $\sigma_n^A = \emptyset$ and $\sigma_n^N \neq \emptyset$ for large n. By ROB-3, we must have $\texttt{ready}_n \neq \emptyset$ then. From $\rho_n = \emptyset$ (implied by $\sigma_{n+1}^{RD} = \sigma_n^{RD}$) we have $\texttt{error}_n = \mathit{false}$, so $\sigma_{n+1}^A \neq \emptyset$ (by proof of Lemma 9), and this contradicts $\omega_{n+1} \in N^*$.

Finally, suppose $X = C$ or $X = E$. By ROB-1, namely the condition $l \geq 1$ there, it follows that $\sigma_{n+1}[k+1] = R$, which is again a contradiction. □

Now we obtain a useful factorization of σ_∞. Recall from Lemma 2 that $\sigma_{n+1}^{RD} = \sigma_n^{RD} \rho_n \delta_n$, where $\mathbf{sts}(\delta_n) \in D^*$, and (1) $\delta_n = \emptyset$ if n is regular, and (2) $\rho_n \neq \emptyset$ if n is singular. Write $\psi_n = \rho_n \delta_n$ and note that $\psi_n \neq \emptyset$ implies $\rho_n \neq \emptyset$. Lemma 13 implies that

$$\sigma_\infty = \psi_1 \psi_2 \cdots = (\rho_1 \delta_1)(\rho_2 \delta_2) \cdots$$

Since σ_∞ is infinite there are infinitely many nonempty factors ψ_n, and each nonempty ψ_n begins with a non-empty ρ_n. Consequently, there is no end to retiring:

Lemma 14 (Retiring). *The sequence ρ_∞ is infinite.* □

The fetching process defines another natural factorization of σ_∞:

$$\sigma_\infty = \phi_1 \phi_2 \cdots,$$

where ϕ_n are simply defined by $|\phi_n| = |\texttt{fetched}_n|$.

Lemma 15. $\langle \mathbf{pc}, \mathbf{spc} \rangle (\sigma_\infty) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\texttt{fetched}_1 \texttt{fetched}_2 \cdots)$.

Proof. It suffices to show $\langle \mathbf{pc}, \mathbf{spc} \rangle (\sigma_{n+1}) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\sigma_n \texttt{fetched}_n)$. Using the ROB axioms and notation again, our claim follows from a sequence of equalities: $\langle \mathbf{pc}, \mathbf{spc} \rangle (\sigma_n \texttt{fetched}_n) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\xi_1) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\xi_2) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\xi_3) = \langle \mathbf{pc}, \mathbf{spc} \rangle (\sigma_{n+1})$. The third equality follows from Corollary 1. The other three follow easily from definitions. □

The two factorizations of σ_∞ are related as follows.

Lemma 16 (Factorizations). *If n is regular, then $\psi_1 \cdots \psi_n$ is a prefix of $\phi_1 \cdots \phi_{n-1}$. If n is singular, then $\psi_1 \cdots \psi_n = \phi_1 \cdots \phi_n$ and δ_n contains ϕ_n.*

Proof. By Lemma 2, $\sigma_{n+1}^{RD} = \sigma_n^{RD} \psi_n$, so $\psi_1 \cdots \psi_n = \sigma_{n+1}^{RD}$. Going back to the transition process analyzed at the beginning of §6, we can now use the information from Lemma 9 to describe the passage from ξ_2 to ξ_3 more precisely. Recall that $\zeta_2 = \tau_n^{RD} \omega A^k N^{p+q-k}$ and $\zeta_3 = \tau_n^{RD} \omega' N^{p+q-k}$, where ω' is obtained from ωA^k by replacing some A's with C's or E's. We claim that these changes in fact occur entirely within ω. The reason is in that the subword A^k of ζ_2 corresponds to $\mathtt{ready}_n$, and the names of transactions in $\mathtt{ready}_n$ do not occur among the names of transactions in $\mathtt{computed}_n$. Indeed, using the proof of Lemma 9 and its notation, we have $\mathbf{name}(\mathtt{computed}_n) \subset \mathbf{name}(\mathtt{contents}_n) \subset \cup_{i<n}\mathbf{name}(\mathtt{ready}_i)$.

Thus, the critical segment of ζ_3 is entirely within the image of τ_n in ζ_3, so the suffix of length $p+q$ of ζ_3 does not intersect the critical segment. If n is regular, that implies $|\tau_{n+1}^{RD}| < |\tau_n| = |\phi_1| + \cdots + |\phi_{n-1}|$. Similarly, if n is singular, it follows that the suffix of length $p+q$ of $\sigma_{n+1} = \sigma_{n+1}^{RD}$ belongs to δ_n. The two statements of the lemma immediately follow. □

4.4 Axioms of the standard machine

Proof of SM-1.

We want to prove $\mathbf{pc}(\rho_\infty[1]) = \mathtt{startIdx}$. Let n be the smallest integer such that $\mathtt{fetched}_n \neq \emptyset$. By IFU-1, $\mathbf{pc}(t) = \mathtt{startIdx}$, where t is the first transaction in $\mathtt{fetched}_n$. In view of Lemma 15, σ_∞ begins with a transaction t' such that $\mathbf{pc}(t) = \mathbf{pc}(t')$, so it suffices to check that t' is in ρ_∞. Indeed, t' is the first transaction in the first non-empty ψ_m, so it belongs to ρ_m (see the paragraph after Lemma 13).

Proof of SM-2.

Since ρ_∞ is infinite, what needs to be checked is that $\mathbf{pc}(t') = \mathtt{next}(\langle\mathbf{pc}, \mathbf{res}\rangle(t))$ for any two consecutive transactions t, t' in ρ_∞. First we consider the case when t is faulty. By Lemma 6, t is the last element of some ρ_n, where n is singular. Thus, $t = \mathtt{lastRet}_n$. By Lemma 16, $\psi_1 \cdots \psi_n = \phi_1 \cdots \phi_n$. Let m be the smallest integer such that $\psi_1 \cdots \psi_n = \phi_1 \cdots \phi_m$ and $\phi_m \neq \emptyset$. We claim that every i such that $m < i \leq n$ is regular. Indeed, if such an i were singular, we would have $\psi_1 \cdots \psi_i = \phi_1 \cdots \phi_i$ (Lemma 16). Since $\phi_1 \cdots \phi_n = \phi_1 \cdots \phi_i$, it would follow that $\psi_1 \cdots \psi_n = \psi_1 \cdots \psi_i$, which is not true since ψ_n is not empty.

As for t', we have that it is the first element of the first non-empty ψ_j that comes after ψ_n. Since $\psi_1 \cdots \psi_n = \phi_1 \cdots \phi_n$, it follows that t' is also the first element of the first non-empty ϕ_k that comes after ϕ_m. Since n is the smallest singular integer among $m, m+1, \ldots, k-1$, it follows from IFU-3b that $\mathbf{pc}(t') = \mathtt{nextpc}(\mathtt{lastRet}_n) = \mathtt{nextpc}(t)$, finishing the proof in the case when t is faulty.

Assume now t is not faulty. Since every transaction of every ρ_n has a corresponding transaction in $\mathtt{computed}_n$ which differs from it only in the status letter (ROB-1,2), it follows from Lemma 6 that $\mathbf{spc}(t) = \mathtt{nextpc}(t)$. This reduces our problem to showing that $\mathbf{pc}(t') = \mathbf{spc}(t)$. If t, t' are consecutive elements of some ϕ_n, this is exactly what we get from IFU-2 with the aid of Lemma 15. Now, we do have that t and t' are consecutive in $\sigma_\infty = (\rho_1\delta_1)(\rho_2\delta_2)\cdots$, because otherwise t would have to be the last element of a ρ_n, where $\delta_n \neq \emptyset$, which would mean that n is singular (Lemma 2), and so by Lemma 6, that t is faulty, which is absurd. Thus, the only remaining case to consider is when t is the last in some ϕ_m and t' is the first in ϕ_n, where $\phi_i = \emptyset$ for i between m and n. The desired result $\mathbf{pc}(t') = \mathbf{spc}(t)$ follows from IFU-3a and Lemma 15 provided $\mathtt{error}_k$ is false for every k such that $m \leq k < n$. this last condition does not necessarily hold, so suppose finally that k is the smallest integer in this interval for which $\texttt{error}_k$ is true. Thus, k is singular (Lemma 7), so $\psi_1 \cdots \psi_k = \phi_1 \cdots \phi_k = \phi_1 \cdots \phi_m$. It follows that t is the last element of $\psi_k = \rho_k \delta_k$, so $\delta_k = \emptyset$ and $t = \texttt{lastRet}_k$. The axiom IFU-3b applies, so $\mathbf{pc}(t') = \texttt{nextpc}(\texttt{lastRet}_n) = \texttt{nextpc}(t)$, finishing the proof.

Proof of SM-3.

As mentioned above, every transaction of ρ_∞ has a corresponding transaction in some $\texttt{computed}_n$ which differs from it only in the status letter. Thus, in view of RS-2, $\mathbf{res}(t) = \texttt{compute}(\langle \mathbf{pc}, \mathbf{ops}\rangle(t))$ holds for every t in ρ_∞. Therefore, to prove that $\langle \mathbf{pc}, \mathbf{res}\rangle(\rho_\infty)$ satisfies SM-3, it suffices only to check that all transactions in ρ_∞ have correct operands. This amounts to the following two lemmas.

Lemma 17. *If $\sigma_\infty[p]$ is the first source transaction of $\sigma_\infty[i]$ in ρ_∞, then p is the first source of i. (Similarly for the second sources.)*

Proof. Suppose $\sigma_\infty[i]$ belongs to ϕ_m and ψ_n. Then $m < n$ and every k such that $m \leq k < n$ is regular. For some m' ($m \leq m' < n$), the set $\texttt{ready}_{m'}$ contains a transaction with name i, and the first source p' of i is determined by this condition: $\xi_1[p']$ is the first transaction source of $\xi_1[i]$ in ξ_1°, where ξ_1 is the first intermediate step in the transition from $\sigma_{m'}$ to $\sigma_{m'+1}$. Since m' is regular, $\mathbf{sts}(\xi_1) = \mathbf{sts}(\sigma_{m'+1})$. The definition of transaction sources depends only on the $\mathbf{pc}$ components, so we obtain (using Corollary 1) that $\sigma_{m'+1}[p']$ is the first transaction source of $\sigma_{m'+1}[i]$ in $\sigma_{m'+1}^\circ$. Since all numbers between m' and n are regular, we obtain, arguing by induction, that $\sigma_k[p']$ is the first transaction source of $\sigma_k[i]$ in σ_k° for every k such that $m' < k \leq n$. Since $\mathbf{sts}(\sigma_n[i]) = R$, and σ_n^{RD} is a prefix of σ_∞, it follows that $\sigma_\infty[p']$ is the first transaction source of $\sigma_\infty[i]$ in $\sigma_\infty^\circ = \rho_\infty$. Thus, $p = p'$, proving the lemma. □

Lemma 18. *If $\sigma_\infty[p]$ and $\sigma_\infty[i]$ are in ρ_∞ and if p is the first source of i, then* $\mathbf{op1}(\sigma_\infty[i]) = \mathbf{res}(\sigma_\infty[p])$. *(Similarly for the second source/operand.)*

Proof. Let m and be such that $S_m\langle p\rangle \in \texttt{computed}_m$ and $S_n\langle p\rangle \in \texttt{computed}_n$. We have $\mathbf{res}(\sigma_\infty[i]) = \mathbf{res}(S_n\langle i\rangle)$ and $\mathbf{ops}(\sigma_\infty[p] = \mathbf{ops}(S_m\langle p\rangle)$, so it suffices to prove that $\mathbf{op1}(S_n\langle i\rangle) = \mathbf{res}(S_m\langle p\rangle)$. If $m < n$, then $\mathbf{res}(\sigma_n[p]) = \mathbf{res}(S_m\langle p\rangle)$ and the result follows from Lemma 11(a). If $m > n$ then $S_n\langle p\rangle$ exists, and by Lemma 11(b), $\mathbf{res}(\sigma_n[p]) = \mathbf{res}(S_n\langle p\rangle)$. Since $\mathbf{res}(S_n\langle p\rangle)$ is in $\texttt{Value}$, it must be equal to $\mathbf{res}(S_m\langle p\rangle)$, and again the desired result follows. □

5 Related work

Burch and Dill's seminal paper [2] developed the concept of a *pipeline flushing* abstraction function to prove an equivalence between an ISA and a pipelined implementation. Any instructions in flight are made to complete by an appropriate insertion of null operations. Since then, Burch [3], Windley and Burch [12], and Skakkebæk, Jones and Dill [11] have extended the approach to superscalar pipelined microprocessors. Using a non-deterministic intermediate machine, Damm and Pnueli [5] constructed a *refinement* relation between a sequential and Tomasulo-style implementation of an out-of-order processor core. McMillan [8] verifies the same processor using *compositional model-checking* techniques. The machine transitions are defined by next-state operations on the full states of the ISA and the Tomasulo machines. This results in a large conjunctive formula, each conjunct of which is checked independently.

Arguably, techniques like these that expose all of the microarchitecture's state do not fit well with hierarchical design methodologies. In developing the proof in this paper, we attempted to hide as much as possible of the local state of each component. The external behavior of each component is constrained by the component's axioms, but within those constraints the component's state is unspecified. This encapsulation may provide an additional level of abstraction and modularity to the verification effort, and allow separate teams to develop each component, while ensuring that global processor invariants are maintained.

Shen and Arvind [10] describe a term-rewriting methodology for verifying superscalar, speculative, out-of-order multiprocessors. Their approach can be considered to be at an even higher level of abstraction than our Hawk designs which provided the basis for our axiomatization. As a result, their specifications are simpler than ours. However, most state transitions are defined across the machine as a whole rather than localized to each component and subject only to the inputs of that component. Furthermore, their model does not contain any explicit clocking mechanism, so it is not clear how to derive cycle-accurate microarchitectural specifications. As their models rely on being able to apply rewrite rules in any order, it is not clear how their correctness result would translate to a lower-level implementation that did not have this flexibility.

In a very recent paper, Sawada and Hunt [9] describe a microprocessor verification that has many similarities to our work. They also construct a sequence of transaction-like records, called a *Micro-Architecture Execution Trace Table* (MAETT). Like our σ state, the MAETT is permanently enlarged with every completed instruction. Each entry in the MAETT stores a unique instruction identifier, all operands and results of the instruction as ISA states, and the pipeline state in which the instruction is currently located, plus other fields. These entries correspond strongly to the transactions we use. The requirements the MAETT must satisfy are given abstractly in terms of the ISA state and the microarchitecture state, in a similar way to our component axioms. However, the structure of their proof also requires them to construct an invariant between successive microarchitectural states. This invariant is likely to reference most of the state elements of the micro-architecture. It is the most difficult construction in their proof.

6 Conclusions

Rather than rely on "flushing" dynamic state to show equality with the ISA state, we define correctness in terms of visible outputs. This means that we can avoid having to demonstrate equivalence between internal states of the ISA and the dynamic machine, so long as the *outcome* of the computations are the same. This may be important in practice because the models constructed from a realistic machine are commonly too big even to construct, let alone verify. By imposing a hierarchical view on the design, we hope to mitigate this problem to some extent.

Our axiomatization can be satisfied by a family of microarchitectures. This means that it retains a good deal of flexibility as the structure of individual components is developed. Each component is specified independent of other components, in the same way RTL design is organized. Once the overall microarchitecture has been developed, the implementation and proof can be carried out independently. Although flaws found during the proof might affect the microarchitectural design, this is true for other analyses such as time and performance estimating. Furthermore, the specification and its correctness proof are independent of many configurations that effect performance. For example, the specification does not explicitly set the latency of the RS and its execution units, the number of execution units, the width of the `computed` wire, or the accuracy of branch prediction. Therefore, many design decisions based on simulation may be made without adversely affecting the global correctness proof.

We intend to repeat this work for a number of related microarchitectural forms, and so to build a framework, based on the concept of signals of transactions, for axiomatizing the major components of dynamic execution machines, and proving global correctness properties. We would identify the most useful axioms for common components and prove consequent lemmas about those components or about typical interactions between them.

We also need to confirm that the axiomatizations can be related to specific microarchitecures. As mentioned earlier, we developed an executable P6-like specification in Hawk using the same structure described here. We plan to prove the correctness of this executable model by proving that the component axioms hold for the specifications of the RS, ROB, and IFU. We foresee a couple of complications in achieving this. First, the Hawk model supports full memory instructions, and these are present in the current work only in a very rudimentary fashion. Secondly, the Hawk model contains extra inter-unit communication for dealing with finite bounds. These facilities need to be included in the axiomatic model before the two can be formally related.

7 Acknowledgments

For their contributions to this research, we thank Mark Aagaard, Borislav Agapiev, Robert Jones, and John O'Leary of Intel Strategic CAD Labs; Tito Autrey and Dick Kieburtz of OGI; and Arvind of MIT.

The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews recieves support from a graduate research fellowship with the National Science Foundation.

References

[1] AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).

[2] BIRCH, J., AND DILL, D. Automatic verification of pipelined microprocessor control. In *6th International Conference of Computer Aided Verification* (Stanford, California, June 1994).

[3] BURCH, J. Techniques for verifying superscalar microprocessors. In *33rd annual Design Automation Conference* (Las Vegas, Nevada, June 1996).

[4] COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

[5] DAMM, W., AND PNUELI, A. Verifying out-of-order executions. In *Conference on Correct Hardware Design and Verification Methods* (Montreal, Canada, 1997).

[6] GWENNAP, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995).

[7] MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Aug. 1998).

[8] MCMILLAN, K. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).

[9] SAWADA, J., AND HUNT, W. Processor verification with precise exceptions and speculative execution. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).

[10] SHEN, X., AND ARVIND. Design and verification of speculative processors. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

[11] SKAKKEBAEK, J., JONES, R., AND DILL, D. Formal verification of out-of-order execution using incremental flushing. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).

[12] WINDLEY, P., AND BURCH, J. Mechanically checking a lemma used in an automatic verification tool. In *Formal Methods in Computer-Aided Design* (Palo Alto, California, 1996).